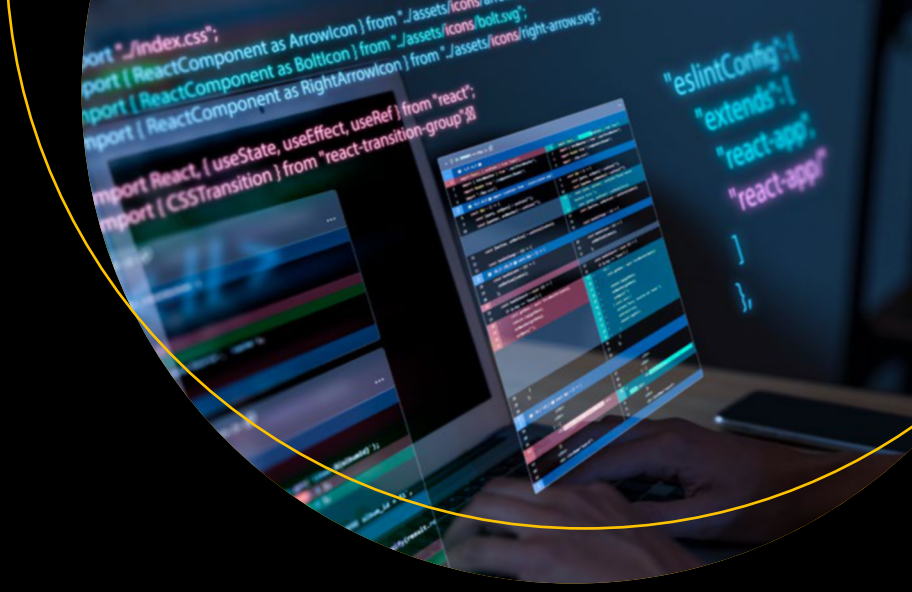

import "./index.css";
import { ReactComponent as ArrowIcon } from "./assets/icons/arrow.svg";
import { ReactComponent as BoltIcon } from "./assets/icons/bolt.svg";
import { ReactComponent as RightArrowIcon } from "./assets/icons/right-arrow.svg";
import React, { useState, useEffect, useRef } from "react";
import { CSSTransition } from "react-transition-group";
"eslintConfig": {
"extends": [
"react-app",
"react-app/"
]
},

# Parallel Programming with C# and .NET

Fundamentals of Concurrency and Asynchrony Behind Fast-Paced Applications

—

Vaskaran Sarcar

*Foreword by Naga Santhosh Reddy Vootukuri*

Apress®

# Parallel Programming with C# and .NET

## Fundamentals of Concurrency and Asynchrony Behind Fast-Paced Applications

**Vaskaran Sarcar**

***Foreword by Naga Santhosh Reddy Vootukuri***

Apress®

*Parallel Programming with C# and .NET: Fundamentals of Concurrency and Asynchrony Behind Fast-Paced Applications*

Vaskaran Sarcar
Near Garia Station, Post: Garia
Kuntala Furniture, 2nd Floor
Kolkata, West Bengal, India

ISBN-13 (pbk): 979-8-8688-0487-8 ISBN-13 (electronic): 979-8-8688-0488-5
https://doi.org/10.1007/979-8-8688-0488-5

Copyright © 2024 by Vaskaran Sarcar

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Laura Berendson
Coordinating Editor: Kripa Joseph

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress/Parallel-Programming-with-CSharp-and-.NET). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

*To those who are often labeled as “workaholics,” “introverted,” or “antisocial,” only because they forget the world while coding.*

# Table of Contents

**About the Author** ..... xi

**About the Technical Reviewer** ..... xiii

**Acknowledgments** ..... xv

**Foreword** ..... xvii

**Introduction** ..... xix

**Chapter 1: Understanding Tasks** ..... 1

Stepping into Parallel Development ..... 1

Introduction to the Task Parallel Library ..... 3

How Does the TPL Help? ..... 4

The Concept of Tasks ..... 4

Creating and Executing a Task ..... 5

Passing and Returning Values ..... 14

Continuation Tasks ..... 20

Simple Continuation ..... 21

Specialized Continuation ..... 23

Nested Tasks ..... 29

Using TaskCreationOptions ..... 31

Using TaskContinuationOptions ..... 33

Discussion of Waiting ..... 42

Why Do We Wait? ..... 43

How Do We Wait? ..... 44

Exercises....57
Summary....61
Solutions to Exercises....61

**Chapter 2: Handling Special Scenarios....71**

Introduction to Exceptions....71
Understanding the Challenge....71
Retrieving the Error Details....76
Exception Management....80
Final Suggestion....91
Understanding Cancellations....91
Different Ways of Cancellation....92
Monitoring Task Cancellation....103
Cancelling Child Tasks....107
Managing Multiple Cancellation Tokens....107
Organizing Exceptions and Cancellations....112
Case Study 1: Using Wait()....112
Case Study 2: Using Wait(token)....115
Handling I/O-Bound Tasks....117
Using TaskCompletionSource....118
Exercises....125
Summary....136
Solutions to Exercises....136

**Chapter 3: Exploring Synchronization and Concurrent Collections....143**

Synchronization....143
Understanding Why We Need Synchronization....145
Using Lock Statements....148

Using Interlocked Classes ..... 150
Signaling Using AutoResetEvent ..... 154
Concurrent Collections ..... 162
System.Collections.Concurrent Namespace ..... 162
IProducerConsumerCollection<T> Interface ..... 163
Notable Characteristics ..... 168
ConcurrentStack<T> ..... 172
ConcurrentQueue<T> ..... 182
ConcurrentBag<T> ..... 185
BlockingCollection<T> ..... 194
Note from Microsoft ..... 201
ConcurrentDictionary<TKey,TValue> ..... 201
Exercises ..... 210
Summary ..... 213
Solutions to Exercises ..... 214

**Chapter 4: Working on Parallel Loops ..... 219**

Revisiting Sequential Loops ..... 219
Experimenting with Parallel Loops ..... 220
Introducing the Parallel Class ..... 221
Parallel.ForEach ..... 222
Parallel.For ..... 223
Parallel.Invoke ..... 226
Scenarios for Parallel Execution ..... 230
Case Study 1 ..... 231
Case Study 2 ..... 232
Analysis ..... 234

Useful Parameters ..... 234
Using ParallelLoopState ..... 235
Using ParallelOptions ..... 241
Managing Cancellations ..... 245
Following the Previous Approach ..... 245
Recommended Approach ..... 246
Handling Exceptions ..... 250
Fine-Tuning ..... 254
Using Thread-Local Variables ..... 256
Using Partition-Local Variables ..... 260
Additional Note ..... 260
Reviewing Different Coding Styles ..... 265
Introducing the ParallelEnumerable Class ..... 266
Exercises ..... 267
Summary ..... 271
Solutions to Exercises ..... 272

**Chapter 5: Parallel LINQ ..... 279**

Prerequisite Knowledge ..... 279
Imperative vs. Declarative Programming ..... 279
PLINQ Supports Declarative Programming ..... 280
Understanding LINQ Is Beneficial ..... 281
The Path Toward Parallelism ..... 281
Converting LINQ into PLINQ Using AsParallel ..... 282
Using the ParallelEnumerable Class ..... 283
Introducing ForAll ..... 284

Getting Familiar with PLINQ ..... 286
The First Program ..... 286
Do Parallel Queries Run Faster Than Sequential Queries? ..... 291
Forcing Parallelism and Controlling the Degree ..... 298
Merging Data ..... 300
Merge Options ..... 300
Managing Special Scenarios ..... 306
Handling Exceptions ..... 306
Handling Cancellations ..... 310
Exercising Aggregation ..... 315
Sequential Custom Aggregation ..... 316
Parallel Custom Aggregation ..... 318
Exercises ..... 322
Summary ..... 325
Solutions to Exercises ..... 326

**Chapter 6: Simplifying Asynchronous Programming ..... 333**

Introduction to async and await ..... 334
Understanding async ..... 335
Understanding await ..... 339
State Machines Are Behind the Scenes ..... 349
Task.Run vs. Task.Factory.StartNew ..... 359
Task.Factory.StartNew Treats Ordinary Lambdas and Asynchronous Lambdas Differently ..... 360
Task.Run Treats Ordinary Lambdas and Asynchronous Lambdas Uniformly ..... 361
Unwrapping Using the Unwrap Method ..... 362
Unwrapping Using await ..... 362

Asynchronous Construction ..... 363
Approach 1 ..... 366
Approach 2 ..... 368
Approach 3 (Using a Static Method aka Factory) ..... 372
Approach 4 (Lazy Initialization) ..... 375
Introducing ValueTasks ..... 383
Consuming ValueTasks ..... 384
Reviewing Exceptions ..... 387
Exercises ..... 390
Summary ..... 393
Solutions to Exercises ..... 394

**Appendix A: Supplementary Notes ..... 403**

**Appendix B: Recommended Reading ..... 443**

**Appendix C: Other Books by the Author ..... 445**

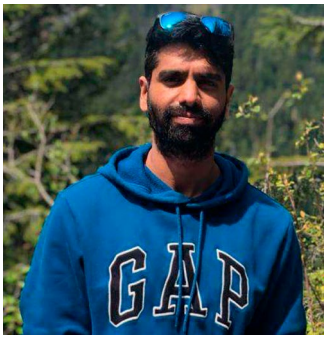
**Index ..... 447**

# About the Author

**Vaskaran Sarcar** obtained his master's of engineering from Jadavpur University, Kolkata (India), and his master's of computer applications from Vidyasagar University, Midnapore (India). He was a National Gate Scholar (2007–2009) and has more than 12 years of experience in education and the IT industry. He devoted his early years (2005–2007) to the teaching profession at various engineering colleges, and later he joined HP India PPS R&D Hub in Bangalore. He worked there for more than 10 years and became a senior software engineer and team lead. After that, he decided to follow his passion and is now an independent full-time author.

You can find his books at https://amazon.com/author/vaskaran_sarcar and find him on LinkedIn at www.linkedin.com/in/vaskaransarcar.

# About the Technical Reviewer




**Naga Santhosh Reddy Vootukuri** is a senior software engineering manager at Microsoft, specializing in cloud computing and artificial intelligence. With a distinguished career spanning 16 years across India, China, and the United States, Naga has amassed a wealth of experience in distributed systems, microservices, and large-scale infrastructure management building cloud and intelligent systems.

At Microsoft, Naga leads the Azure SQL Database team, driving initiatives to optimize SQL deployment processes for millions of databases worldwide. His leadership was pivotal in the development of Master Data Services (MDS), a critical component of Microsoft's SQL enterprise solutions, which saw a significant increase in customer adoption under his stewardship.

Naga has authored and published numerous research articles in peer-reviewed and indexed journals. He is a senior member of IEEE, teaches workshops on AI, and writes technical articles as a Core MVB member at DZone, engaging with millions of active readers. You can read his articles at `https://dzone.com/authors/sunnynagavo`. He also serves as an editorial board member for a highly reputed science journal (SCI), where he reviews research articles on cloud computing and AI, further solidifying his influence in the academic and professional spheres.

Beyond his corporate roles, Naga actively contributes to the tech community by speaking at events, reviewing books for Apress, and actively helping people in forums like the Microsoft Tech Community. His commitment to advancing technology is underscored by his recent role as a judge for the Globee Awards, where he evaluated and recognized innovations in the industry. He also served as a judge for the Microsoft AI hackathon, with more than 10,000 developers participating.

Throughout his career, Naga has demonstrated exceptional leadership and technical prowess, evident in his mentorship on ADP List and his key role in conducting more than 100+ interviews to recruit top talent for Microsoft. You can contact him at https://adplist.org/mentors/sunny.

In summary, Naga embodies a blend of technical expertise, leadership excellence, and community engagement, making profound contributions to both Microsoft and the broader tech industry. His dedication to innovation and mentorship continues to inspire professionals and shape the future of technology globally.

# Acknowledgments

First, I thank the Almighty. I sincerely believe that with His blessings only I could complete this book. I also extend my deepest gratitude and thanks to the following:

- **Naga Santhosh Reddy Vootukuri:** He is the technical reviewer of this book, located in a different country, and works in a different time zone. Despite these factors, whenever I was in need, he provided support. He answered all my queries through phone calls, WhatsApp, and emails. Thank you one more time.
- **Smriti, Laura, Celestin, and the Apress team:** I sincerely thank each of you for giving me another opportunity to work with you and with Apress.
- **Nirmal, Kim Wimpsett, Selvakumar and Pushparaj:** Thanks to each of you for your exceptional support in beautifying my work. Your efforts are extraordinary.

Finally, I thank those people from the online C# developer community, .NET developer community, and Stack Overflow community who have shared their knowledge in various forms. In fact, I thank everyone who directly or indirectly contributed to this work.

# Foreword

Mastering a programming language is much like an artist perfecting their craft; it demands dedication, practice, and the right tools. In the realm of modern computing, where efficiency and speed are paramount, parallel programming stands as a cornerstone. In fact, parallel programming is like conducting an orchestra—each thread must play its part perfectly, or you end up with chaos instead of harmony. This book, *Parallel Programming with C# and .NET*, is an indispensable guide for anyone looking to harness the full power of parallel computing, ensuring their applications are both fast and scalable.

Vaskaran Sarcar, an esteemed author, is known for his ability to explain complex ideas clearly. He is passionate, well-informed, skilled, and very knowledgeable in this area. His systematic approach and curiosity have earned him respect in the programming community. I've seen Vaskaran tackle tough problems with focused determination, consistently finding elegant solutions that showcase his analytical skills and dedication.

*Parallel Programming with C# and .NET* focuses sharply on the complex world of parallel computing. Each chapter breaks down different techniques step-by-step, starting with basic concepts explained in a simple manner. It includes practical sample programs and quizzes to reinforce your understanding. This structured approach simplifies learning and ensures you can apply these concepts effectively. This pedagogical style ensures that readers not only grasp fundamental concepts but also retain them effectively.

In today's tech landscape, performance and efficiency are crucial. C# has grown beyond its Windows origins, now used in diverse environments like Mac and Linux, thanks to Microsoft's open-source culture. This book provides advanced knowledge to help developers fully utilize C#'s capabilities, crafting efficient solutions optimized for various platforms.

I am confident that *Parallel Programming with C# and .NET* will accelerate many developers' learning so they become proficient in parallel computing in record time. Vaskaran's exemplary work establishes a solid foundation for understanding and utilizing advanced language features, paving the way for creating robust software.

Developers will gain clarity on complex concepts and rely on this book as their essential resource for mastering parallel computing challenges. Happy coding!

Naga Santhosh Reddy Vootukuri
Senior Software Engineering Manager
Microsoft, Azure SQL Server
(Cloud + AI division)

# Introduction

Modern-day software is highly responsive and scalable. As a result, support for parallel computation is an essential part of it. Undoubtedly this is an advanced concept, and the solutions are not straightforward. Many developers have been burned (and are still burning) by them.

In addition, parallel programming is a vast topic that requires many different considerations. You may also note that many patterns used in the past to deal with asynchronous and parallel programming are not recommended now. This book tries to simplify the concept using modern C# features and libraries that Microsoft recommends. Welcome to your journey through *Parallel Programming with C# and .NET: Fundamentals of the Concurrency and Asynchrony Behind Fast-Paced Applications*.

C# is a powerful programming language, well-accepted in the programming world, that can help you make a wide range of applications. Throughout C# development, supportive features and libraries have been developed to support parallel programming. This is one of the primary reasons that it is continuously growing and always in high demand. So, it is not a surprise that existing and upcoming developers (for example, college students and programming lovers) want to use C# for parallel programming.

Many developers try to learn parallel programming in the shortest possible time frame. Trying to learn something as quickly as possible is laudable, but do you know the problem? We are living in a world that offers you lots of materials, advertisements, and quick fixes to capture your attention. We human beings love to daydream. So, they take advantage of this behavior and start claiming that you can learn everything in a day, a week, or a month. Is this true? Ask yourself and you'll get the answer.

Malcolm Gladwell in his book *Outliers* (Little, Brown, and Company) talked about the 10,000-hour rule. This rule says that the key to achieving world-class expertise in any skill is, to a large extent, a matter of practicing the correct way for a total of about 10,000 hours. So, you can probably see that even though we may claim that we know something very well, we actually probably know very little. Learning is a continuous process, with no end.

Then should we stop learning? Definitely, the answer is no. What should we do then? We can follow an *effective learning* process that teaches you how to learn quickly to serve the need. This is where I like to remind you about the Pareto principle, or 80-20 rule. This rule simply states that 80% of outcomes come from 20% of all causes. This is useful in programming too. When you truly learn the fundamental and most important aspects of parallel programming, that is when you can use it effectively to improve your code. Most importantly, your confidence level will rise, and you won't be afraid to experiment more. This book is for those who acknowledge these facts.

## How Is the Book Organized?

This book helps you to understand the core principles of parallel programming using six chapters with plenty of supportive materials. To give you an idea how each chapter is organized, the following list talks about the contents of the book:

- Chapter 1 starts with an overview of the Task Parallel Library (TPL) and discusses tasks. These topics are the foundation for the upcoming chapters. Chapter 2 discusses special scenarios such as handling exceptions and cancellations. Chapter 3 discusses synchronization techniques and concurrent collections. Chapter 4 discusses the `Parallel` class, which helps you experiment with parallel loops to speed up

computations. Chapter 5 discusses Parallel LINQ (PLINQ). Chapter 6 discusses simplifying asynchronous programming using the `async` and `await` keywords. Finally, the appendix provides you with some extra materials that are not discussed in the previous chapters.

- I have always enjoyed learning when analyzing case studies, asking questions, and performing exercises. So, throughout this book, you will see interesting program code, "Q&A Sessions," and exercises. By analyzing these Q&As and doing the exercises, you can verify your progress. As mentioned, these are presented to make your learning easier and more enjoyable, but most importantly, they will make you confident as a developer.
- Each question in these "Q&A Sessions" is marked with **Q<chapter#>.<question#>**. For example, Q5.3 means question 3 from Chapter 5. At the end of the chapter, you'll see some exercises. You can use them to evaluate your progress. Each question in these exercises is marked with **E<chapter#>.<question#>**. For example, E6.2 means exercise 2 from Chapter 6.
- You can download all the source code for the book from the publisher's website.

# Prerequisite Knowledge

I expect you to be very familiar with C#. In fact, knowing about some of the advanced concepts like delegates and lambda expressions can accelerate your learning. So, I assume that you know how to compile or run a C# application in Visual Studio. This book does not invest time in basic

topics, such as how to install Visual Studio on your system, how to write a "Hello World" program in C#, and so forth. In short, the target readers of this book are those who want to make the most of C# by harnessing the power of both object-oriented programming (OOP) and functional programming (FP).

## Who This Book Is For

You will get the most from this book if you can answer "yes" to the following questions:

- Are you familiar with .NET, C#, and basic object-oriented concepts such as polymorphism, inheritance, abstraction, and encapsulation?
- Are you familiar with some of the advanced concepts in C# such as delegates, lambda expressions, and generics?
- Do you know how to set up your coding environment?
- Are you interested in knowing how the modern-day constructs of C# can help you in parallel programming?

You will probably struggle with this book if you can answer "yes" to any of the following questions:

- Are you looking for a C# tutorial or reference book?
- Are you not ready to experiment with parallel programming using a programming language other than C#?
- Do you dislike Windows, Visual Studio, and/or .NET or want to learn parallel programming without them?

# Useful Software

These are the important tools that I used for this book:

- All the programs were tested with C# 12 and .NET 8. In this context, it is useful to know that nowadays the C# language version is automatically selected based on your project's target framework so that you can always get the highest compatible version by default. In the latest versions, Visual Studio doesn't support the UI to change the value, but you can change it by editing the `.csproj` file.
- As per the new rule, C# 12 is supported only on .NET 8 and newer versions. C# 11 is supported only on .NET 7 and newer versions. C# 10 is supported only on .NET 6 and newer versions. If you are interested in learning more about the C# language versioning, you can refer to https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version.
- During the development of this book, software updates kept coming and I also kept updating. When I finished my initial draft, I had the latest edition of Visual Studio Community 2022 (64-bit, version 17.8.5).
- The good news for you is that this community edition is free. If you do not use the Windows operating system, you can use Visual Studio Code, which is also a source-code editor developed by Microsoft that runs on Windows, macOS, and Linux operating systems.

> This multiplatform integrated development environment (IDE) is also free. However, I recommend that you check the license and privacy statement as well because this statement may change in the future.

**Author's Note:** I have tested my code only on Visual Studio. At the time of this writing, Visual Studio 2022 for Mac is scheduled for retirement on August 31, 2024. To learn more about this, you can refer to https://learn.microsoft.com/en-us/visualstudio/mac/what-happened-to-vs-for-mac?view=vsmac-2022.

# Guidelines for Using This Book

Here are some suggestions so that you can get the most out of this book:

- This book suits you best if you are familiar with some advanced features in C# such as delegates and lambda expressions. If not, please read about those topics before you start reading this book.
- I organized the chapters in a manner that can help you understand parallel programming. For example, if you directly start using the `async` and `await` keywords in your programs, you may miss many other concepts that actually caused them to appear in C# in a later version. This is why I believe you should read the chapters sequentially. Another reason for this suggestion is that some useful topics may have already been discussed in a previous chapter, and I have not repeated those discussions in the later chapters.
- The programs in this book should give you the expected output in the upcoming versions of C#/Visual Studio. Though I believe that these results should not vary in

other environments, you know the nature of software: it is naughty. So, I recommend that if you want to see the same output, you should mimic the same environment that I am using.

- You can download and install the Visual Studio IDE from `https://visualstudio.microsoft.com/downloads/`. You are expected to get the screen shown in Figure FM-1.

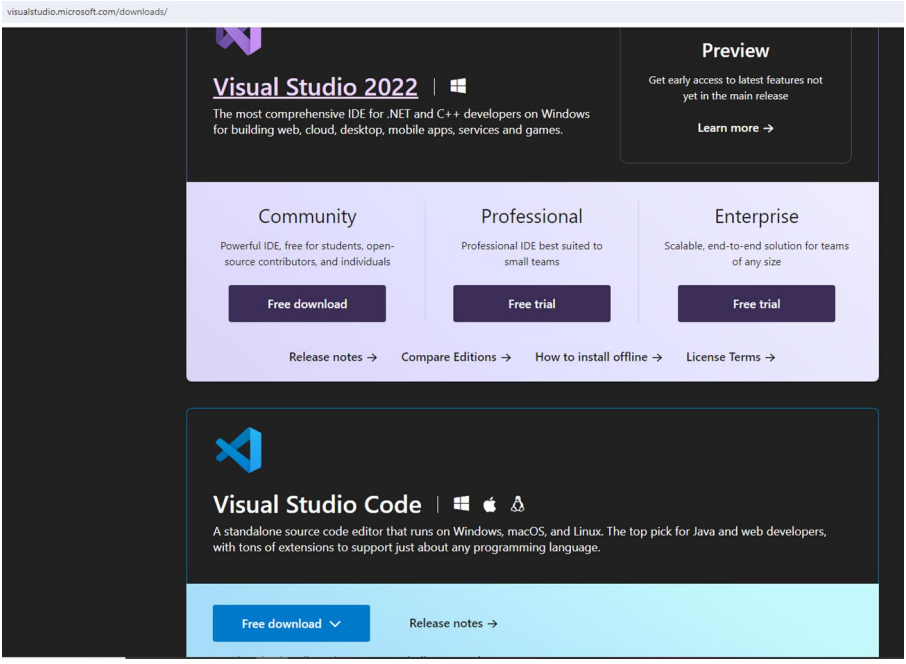

visualstudio.microsoft.com/downloads/
Visual Studio 2022
The most comprehensive IDE for .NET and C++ developers on Windows for building web, cloud, desktop, mobile apps, services and games.
Preview
Get early access to latest features not yet in the main release
Learn more →
Community
Powerful IDE, free for students, open-source contributors, and individuals
Free download
Professional
Professional IDE best suited to small teams
Free trial
Enterprise
Scalable, end-to-end solution for teams of any size
Free trial
Release notes →
Compare Editions →
How to install offline →
License Terms →
Visual Studio Code
A standalone source code editor that runs on Windows, macOS, and Linux. The top pick for Java and web developers, with tons of extensions to support just about any programming language.
Free download
Release notes →

***Figure FM-1.*** *Download screen for Visual Studio 2022 and Visual Studio Code*

**Note** At the time of this writing, this link works fine, and the information is correct. However, the link and policies may change in the future. The same comment applies to all the mentioned links in this book.

# Conventions Used in This Book

In many places, I give you links to Microsoft's documentation. Why? For me, the creators of the products are the authenticated source of information to describe a feature.

In addition, to draw your attention to some places in the code, I have made them bold. For example, consider the following code fragment (taken from Chapter 1 where I discuss continuation tasks and want to draw your attention to a particular line):

```
// Using C#12's "Collection expression" feature
var task3 = Task.Factory.ContinueWhenAll(
  [task1, task2],
  tasks =>
  {
    WriteLine("Arranging dinner.");
  }
 );
```

# Final Words

You are an intelligent person. You have chosen a topic that can assist you throughout your career. As you learn and review these concepts, I suggest you write your own code; only then will you master this area. There is no shortcut for this.

Do you know the ancient story of Euclid and Ptolemy, ruler of Egypt? Euclid's approach to mathematics was based on logical reasoning and rigorous proofs, and Ptolemy asked Euclid if there was an easier way to learn mathematics. Euclid's reply to the ruler? "There is no royal road to geometry."

Though you are not studying geometry, the essence of this reply applies here. You must study these concepts and code. Do not give up when you face challenges. They are the indicators that you are growing better.

**Errata:** I have tried my best to ensure the accuracy of the content. However, mistakes can happen. So, I plan to maintain "errata," and if required, I will make some updates/announcements there. You can visit the online link: https://github.com/Apress/Parallel-Programming-with-CSharp-and-.NET to receive any important corrections or updates.

**An appeal:** You probably understand that good-quality work takes many days and months (even years!) to complete. Many authors like me invest most of their time in writing and heavily depend on it. You can encourage and help these authors by preventing piracy. If you come across any illegal copies of our works in any form on the Internet, I would be grateful if you would provide me or the Apress team with the location using the following link: https://www.apress.com/gp/services/rights-permission/piracy.

**Share your feedback:** I believe that this book is designed for you in such a way that you will develop a keen knowledge of parallel programming using C# and .NET. If you value the effort, I request you provide your valuable feedback on the Amazon review page or any other platform you like.

CHAPTER 1

# Understanding Tasks

Modern-day life is full of work. Consider those working mothers who need to complete a lot of housework before they go to the office. You may see them performing all these activities asynchronously. For example, they start preparing breakfast, and they come back from the kitchen to prepare the kids for school. Then they go back to the kitchen to check the status of the food only to leave the kitchen to start preparing themselves for the office. You can see that with this approach, when they start a task, they do not wait for that task to complete; instead, they start the next task and then go back to check the status of the previous task. This process continues until all these tasks are finished. Psychologists who vote for doing one task at a time may not like this approach, but at the same time, we cannot ignore that this is how people do things. The programming world tries to mimic real-world scenarios. Throughout this book, we will explore those possibilities using C# and .NET.

## Stepping into Parallel Development

You are probably familiar with synchronous method calls where you see that the caller method is blocked until the callee method finishes its execution. To illustrate, consider the following code segment:

```
// Some code before
int temp = RetrieveSomeValue(); // Line 1
int result = Process(temp);// Line 2
// Some code after
```

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5_1

If you exercise this code in a single-threaded environment, you cannot execute line 2 before you get the `temp` value in line 1. Why? Calling the `RetrieveSomeValue` method blocks the thread until the method finishes its execution. We are probably all familiar with this type of coding. In fact, we often exercise typical .NET calls that are blocking.

However, it is not a good idea to always rely on the blocking calls. For example, consider the situation when you trigger a method that attempts to download a large file. While the download is in progress, if you cannot do any other work, that is a problem for sure. What is the solution? You guessed it: you will opt for a multithreading environment and opt for asynchronous operations.

You may also notice that computing machines are becoming faster by the day. The role of multicore CPUs is inevitable. So, regardless of the programming languages, developers will want to take advantage of them. This is why parallel programming is becoming an essential technique for developing powerful software that is highly responsive.

Parallel programming is interesting as well as challenging. Undoubtedly it is hard, but in the past it was harder. In those days, developers had the following options:

- Creating and using threads directly
- Using the `ThreadPool` class
- Using event-based asynchrony
- Using the `AsyncResult` pattern
- Using the callback methods

These are not the recommended approaches now. I'd like to start the discussion with tasks that make up the foundation of the Task Parallel Library (TPL). Why? Once .NET Framework 4 was released, Microsoft stated the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming):

*TPL is the preferred API for writing multithreaded, asynchronous, and parallel code in .NET*

---

**Note** It is also interesting to know that behind the scenes, tasks are queued to the `ThreadPool` class, which determines and adjusts the number of threads. The `ThreadPool` class follows some algorithms to control the load balancing to gain maximum throughput.

---

# Introduction to the Task Parallel Library

Why is the TPL important? As mentioned, handling concurrent programs and adding parallelism to an application are challenging tasks. Undoubtedly, these are advanced concepts of programming. The TPL aims to make them easy for you by providing a set of public types and APIs. You can find them in the following namespaces:

- `System.Threading`
- `System.Threading.Tasks`

**Author's Note:** Developers often refer to the `Parallel` class with the task parallelism constructs as the TPL. You'll learn about the `Parallel` class in Chapter 4. However, at this stage, you do not need to worry about them. You'll learn about these constructs one at a time.

## How Does the TPL Help?

Microsoft's documentation states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl):

> *The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications.*

These are some of the useful scenarios for the TPL:

- Managing a multithreaded environment efficiently
- Scaling the concurrency level
- Providing support for task cancellations
- Managing state
- Partitioning your work

You can easily guess that we are going to cover all these scenarios. For now, it will be sufficient for you to understand that the TPL simplifies the multithreading scenarios and helps you write high-performance code without worrying about the nitty-gritty of threading or other low-level details.

# The Concept of Tasks

What are tasks? Microsoft states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming):

> *The Task Parallel Library (TPL) is based on the concept of a task, which represents an asynchronous operation. In some ways, a task resembles a thread or ThreadPool work item but at a higher level of abstraction. The term task parallelism refers to one or more independent tasks running concurrently.*

A *task* is a code block that represents a unit of work. You use tasks to inform the scheduler that this code block can execute on a separate thread while the main thread of execution continues. For example, consider the following code segment:

```
static void PrintNumbers()
{
    WriteLine("Starts executing the task1.");
    // Doing Some work
    Thread.Sleep(5);
    WriteLine("The task1 is finished.");
}
```

This can be a unit of work, and we can execute it on a separate thread. Some common examples of tasks include the following:

- Perform a calculation and display the result.
- Compute a value with/without a supplied input.
- Ask for a network resource.
- Check the health of a website, for example, pinging a website.

## Creating and Executing a Task

You can create and execute tasks in different ways. Let me show you a sample code segment that creates and starts executing the task:

```
Task task1=new Task(PrintNumbers);
task1.Start();
```

From C# 9 onward, you can use target-typed new expressions. So, the previous code can be further simplified as follows:

```
Task task1=new (PrintNumbers);
task1.Start();
```

Basically, you create a task by providing a delegate that encapsulates the intended code. This delegate can be expressed as a named delegate, an anonymous method, or a lambda expression. This is why I present in the following task example, I encapsulate the necessary code inside a lambda expression:

```
Task task1 = new(
    () =>
    {
       WriteLine("Starts executing the task1.");
       // Simulating a delay
       Thread.Sleep(1);
       WriteLine("The task1 is finished.");
    }
);
```

Now I can start this task as follows:

```
task1.Start();
```

---

**Note** I have introduced the delay using the `Thread.Sleep` method. Later, you'll also see me using `Task.Delay` in a similar context, particularly when I discuss asynchronous programming in more depth in Chapter 6.

---

## The Alternative Approaches

I told you that you can create and execute tasks in different ways. Here I present you with two more approaches.

**Approach 2:** You can create and execute a task in a single operation using `Task.Run` method as follows:

```
Task task2 = Task.Run(
    () =>
    {
        WriteLine("Starts executing the task2.");
        // Simulating a delay
        Thread.Sleep(1);
        WriteLine("The task2 is completed.");
    }
);
```

**Alternative 3:** You can also create and execute tasks in a single operation using `TaskFactory.StartNew` method. The following code segment demonstrates the usage:

```
Task task3 =Task.Factory.StartNew(
    () =>
    {
        WriteLine("Starts executing the task3.");
        // Simulating a delay
        Thread.Sleep(1);
        WriteLine("The task3 is completed.");
    }
);
```

You can choose the approach that suits you best.

**Note** The previous approaches that **explicitly** creates and execute tasks. There are other approaches as well. For example, you can **implicitly** create and execute tasks using **Parallel.Invoke** method. In addition, **TaskCompletionSource<TResult>** class also helps you create specialized tasks that are suitable for particular scenarios. However, let us learn the concepts one at a time. The code examples and exercises in this book use all these methods to create and execute tasks to make you familiar with them.

## Demonstration 1

The following program demonstrates the creation and execution of three different tasks on three different threads. Here are two notable characteristics:

- You'll see three different approaches for task creation and execution that I just discussed.
- In addition, this example also includes the following lines to ensure that these tasks finish their job before the console mode application ends.

```
task1.Wait();
task2.Wait();
task3.Wait();
```

Let's see the complete program now.

**POINT TO REMEMBER**

From .NET 6 onwards, you may notice the presence of implicit global directives for new C# projects. This helps you use the types in these namespaces without specifying the fully qualified names or manually adding a "using directive." You can learn more about this at https://learn.microsoft.com/en-us/dotnet/core/project-sdk/overview#implicit-using-directives.

For the C# projects in this book, I did not change the default settings. As a result, you will not see me mentioning the following namespaces that were available by default:

```
System
System.Collections.Generic
System.IO
System.Linq
System.Net.Http
System.Threading
System.Threading.Tasks
```

```
using static System.Console;

WriteLine("The main thread starts executing.");

// Approach-1
Task task1 = new(
    () =>
    {
        WriteLine("Task-1 starts.");
        for (int i = 100; i < 105; i++)
        {
            Write($"Task-1 prints {i}\t");
            // Simulating a delay
```

```
            Thread.Sleep(1);
        }
        WriteLine("Task-1 is completed.");
    }
);
// Starting the task
task1.Start();

// Approach-2
Task task2 = Task.Run(
    () =>
    {
        WriteLine("Task-2 starts.");
        for (int i = 210; i < 215; i++)
        {
            Write($"Task-2 prints {i}\t");
            // Simulating a delay
            Thread.Sleep(1);
        }
        WriteLine("Task-2 is completed.");
    }
);

// Approach-3
Task task3 = Task.Factory.StartNew(
    () =>
    {
        WriteLine("Task-3 starts.");
        for (int i = 320; i < 325; i++)
        {
            Write($"Task-3 prints {i}\t");
            // Simulating a delay
            Thread.Sleep(1);
```

```
        }
        WriteLine("Task-3 is completed.");
    }
);

WriteLine($"The main thread is doing some other work...");
Thread.Sleep(10);

WriteLine($"Main thread is completed.");
task1.Wait();
task2.Wait();
task3.Wait();
```

## Output

Here is some sample output (the output may vary on your system). You can see that there is a nice mixture of output from all the different threads/tasks.

```
The main thread starts executing.
The main thread is doing some other work...
Task-2 starts.
Task-2 prints 210       Task-1 starts.
Task-1 prints 100       Task-3 starts.
Task-3 prints 320       Task-2 prints 211       Task-2 prints
212       Task-3 prints 321       Task-1 prints 101      Task-3
prints 322        Task-1 prints 102       Task-2 prints
213       Task-2 prints 214       Task-1 prints 103      Task-3
prints 323        Main thread is completed.
Task-3 prints 324       Task-1 prints 104       Task-2 is
completed.
Task-1 is completed.
Task-3 is completed.
```

## Q&A Session

**Q1.1 What are the benefits of using tasks over threads?**

Here are some common benefits:

- The tasks are relatively lightweight. They will help you achieve **fine-grained parallelism**.
- Later you'll see that by using the built-in API for tasks, you can easily exercise useful operations such as waiting, cancellations, continuations, custom scheduling, or robust exception handling. So, when you opt for tasks instead of threads, you'll have **more programmatic control**.

**Q1.2 To create and execute tasks, you have shown me the use of the `Run`, `Start`, and `StartNew` methods. How can I decide which one is best for me?**

If you see the method definitions in Visual Studio, you will see the following.

The `Start` method starts `System.Threading.Tasks.Task`, scheduling it for execution to the specified `System.Threading.Tasks.TaskScheduler`. The `Run` method queues the specified work to run on the thread pool and returns a `System.Threading.Tasks.Task` object that represents that work. This is a lightweight alternative to the `StartNew` method. It helps you start a task with the default values. This indicates that the `Run` method uses the default task scheduler, regardless of a task scheduler that is associated with the current thread. This is why Microsoft provides the following suggestions (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming):

- The `Run` methods are the preferred way to create and start tasks when more control over the creation and scheduling of the task isn't needed.

Microsoft further says that you can use the `StartNew()` method for the following situations:

- Creation and scheduling don't have to be separated, and you require additional task creation options or the use of a specific scheduler.
- You need to pass an additional state into the task that you can retrieve through its `Task.AsyncState` property.

**POINT TO NOTE**

Shortly, you'll learn about child tasks (or nested tasks). You'll learn that by using `TaskCreationOptions.AttachedToParent`, you can attach a child task to the parent task (if the parent task allows this activity). This option is available in some of the overloads of the `StartNew` method. Here is such an overload:

```
public Task StartNew(
 Action action,
 CancellationToken cancellationToken,
 TaskCreationOptions creationOptions,
 TaskScheduler scheduler)
  {
    // Some code
  }
And you can use it as follows:
var printHelloTask = Task.Factory.StartNew(
    () =>
    {
      WriteLine("Hello!");
```

```
        },
        CancellationToken.None,
        TaskCreationOptions.AttachedToParent,
        TaskScheduler.Default
    );
```

But, in case you use the `Run` method, a similar option is not available for you.

# Passing and Returning Values

In this section, I'll discuss how you can pass values to a task or get back a computed value from a task.

## Passing Values into Tasks

Let's start the discussion on passing values into tasks. Consider the following code:

```
static void PrintNumbers(int limit)
{
    for (int i = 0; i < limit; i++)
    {
        Write($"PrintNumbers prints {i}\n");
        // Doing remaining things, if any
        Thread.Sleep(1);
    }
}
```

If you want to execute this method on a separate thread, you need to pass a valid argument for the `limit` parameter from the calling thread. How can you do that? You can use the following lines of codes:

```
var task1 = new Task(() => PrintNumbers(10));
task1.Start();
```

Alternatively, you can use the following line:

```
var task2=Task.Factory.StartNew(() => PrintNumbers(10));
```

or the following line:

```
var task3 = Task.Run(() => PrintNumbers(10));
```

Notice that in either case, you pass the lambda expression:

```
() => PrintNumbers(10)
```

What does this indicate? This lambda does not take any argument, but I pass the required argument explicitly.

Let's investigate an alternative approach. At the time of this writing, the `Task` class has the constructors in Figure 1-1.

public class Task : IAsyncResult, IDisposable
{
public Task(Action action);
public Task(Action action, CancellationToken cancellationToken);
public Task(Action action, TaskCreationOptions creationOptions);
public Task(Action<object?> action, object? state);
public Task(Action action, CancellationToken cancellationToken, TaskCreationOptions creationOptions);
public Task(Action<object?> action, object? state, CancellationToken cancellationToken);
public Task(Action<object?> action, object? state, TaskCreationOptions creationOptions);
public Task(Action<object?> action, object? state, CancellationToken cancellationToken, TaskCreationOpt

***Figure 1-1.*** *The overloaded versions of the Task constructors*

Notice the constructor that is highlighted:

```
public Task(Action<object?> action, object? state);
```

In the same way, you can investigate the `StartNew` method too. At the time of this writing, the `StartNew` method has 16 overloads in the `TaskFactory` class, and one of them is as follows:

```
public Task StartNew(Action<object?> action, object? state)
{
  // Remaining code not shown
}
```

The previous two code segments give you the idea that you can pass an `object` argument to them. So, let me introduce another function called `PrintNumbersVersion2` that takes an `object` parameter and does a similar thing. It is as follows:

```
static void PrintNumbersVersion2(object? state)
{
    int limit = Convert.ToInt32(state);
    for (int i = 0; i < limit; i++)
    {
        Write($"PrintNumbers prints {i}\n");
        // Doing remaining things, if any
        Thread.Sleep(1);
    }
}
```

This time you can write the following:

```
var task3 = new Task(PrintNumbersVersion2,10);
task3.Start();
```

Or the following:

```
var task4 = Task.Factory.StartNew(PrintNumbersVersion2,10);
```

You have now seen five different approaches while passing a state so far. Let's summarize them:

```
// Approach-1:
var task1 = new Task(() => PrintNumbers(10));
task1.Start();

// Approach-2:
var task2=Task.Factory.StartNew(() => PrintNumbers(10));
```

```
// Approach-3:
var task3 = Task.Run(() => PrintNumbers(10));

// Approach-4:
var task4 = new Task(PrintNumbersVersion2,10);
task4.Start();

// Approach-5:
var task5 = Task.Factory.StartNew(PrintNumbersVersion2,10);
```

You can see that in each approach, I passed an `int`. But, notice that in the last two cases, the target method (`PrintNumbersVersion2`) expected an `object`. As a result, these two approaches suffer from the impact of boxing and unboxing. On the contrary, they look cleaner compared to approaches 1, 2, or 3. In the end, it is up to you how you want to organize it.

---

**Note** You can download the project `Demo_PassingValues` to experience the different approaches that you have seen up until now.

---

## Returning Values from Tasks

When you execute a task, you may need to access the final value. In such cases, you need to use the generic version of the `Task` class and the `Result` property. Here is a sample:

```
var task1 = Task<int>.Factory.StartNew(() => 100);
var result = task1.Result;
WriteLine(result);
```

However, Visual Studio will help you recognize that the type argument can be inferred, and as a result, you can further simplify this code as follows:

```
var task1 = Task.Factory.StartNew(() => 100);
var result = task1.Result;
WriteLine(result);
```

Similarly, given the following method:

```
static int Add(int number1, int number2) => number1 + number2;
```

you can write the following:

```
var task2 = Task.Factory.StartNew(() => Add(5, 7));
var result2 = task2.Result;
WriteLine(result2);
```

Let's see a complete program where you deal with some tasks, pass some values into them, and finally retrieve the computed result.

## Demonstration 2

In the following demonstration, I create two tasks where the first task calculates the factorial of an integer (5) and the second task adds two integers (25 and 17). Once these tasks are finished, I retrieve the computed results and display them in the console window. Let's see the complete program now.

---

**Note** I could use `Task.Run` in this demonstration as well. However, from Chapter 2 onward, I'll use `Task.Run` heavily in this book. Before that, I am intentionally showing a few demonstrations using `Task.Factory.StartNew` in this chapter. It is because I want you to be familiar with both of them.

---

```
using static System.Console;

WriteLine("Passing and returning values by executing tasks.");
static string CalculateFactorial(int number)
{
    int temp = Enumerable
                .Range(1, number)
                .Aggregate((x, y) => x * y);
    return $"The factorial of {number} is {temp}";
}

static int Add(int number1, int number2) => number1 + number2;

var task1 = Task.Factory.StartNew(() => CalculateFactorial(5));
var task2 = Task.Factory.StartNew(() => Add(25, 17));

var result1 = task1.Result;
WriteLine(result1);
var result2 = task2.Result;
WriteLine($"The sum of 25 and 17 is {result2}");
WriteLine($"The main thread is completed.");
```

## Output

Here is the output:

```
Passing and returning values by executing tasks.
The factorial of 5 is 120
The sum of 25 and 17 is 42
The main thread is completed.
```

## Q&A Session

**Q1.3 I can see that in the previous demonstration, you did not use the lines `task1.Wait();` and `task2.Wait();`. Was this intentional?**

When you want to get a result from a task, you need to wait until the task finishes its execution. It means you need to invoke a blocking operation. Using the `Result` property, I did the same: I blocked the calling thread until those tasks were finished. As a result, I did not need to use the line `task1.Wait()` or `task2.Wait()` separately.

# Continuation Tasks

Suppose there are two tasks called Task-A and Task-B. If you want to start executing Task-B only after Task-A, you'd probably like to use callbacks. But the TPL makes it easy. It provides the functionality through a continuation task, which is just an asynchronous task. The idea is the same: once an antecedent task finishes, it invokes the next task that you want to continue. **In our example, Task-A is the antecedent task, and Task-B is the continuation task.**

You understand that continuation is just chaining tasks. This is helpful when you want to pass data (or, execute some logic) from an antecedent to the continuation task. The TPL provides many built-in supports in this context.

- You can invoke a single as well as multiple continuation tasks.
- You can control the continuation. For example, if there are three tasks, called Task-A, Task-B, and Task-C, you can decide that Task-C should continue only after both Task-A and Task-B finish their executions. Alternatively, you may decide that Task-C should not wait for both Task-A and Task-B; instead, it should continue when any of them finish the execution.

- You can pass data as well as exceptions to the continuation task.
- You can also cancel a continuation task if you want. This is often useful during an emergency or when you find a typical bug that keeps occurring during the execution of an application.

## Simple Continuation

Assume that a person wants to invite his friends for dinner. At a high level, let's divide the overall activity into three different tasks as follows:

1. Order food.
2. Invite friends.
3. Arrange dinner.

Let's start with a simple continuation scenario where the host decides to do these steps in order, with ordering food at the beginning. Then he invites his friends, and finally, he arranges the dinner. Since the process starts with ordering food, you will see the following task at the beginning:

```
var task1 =
 Task.Factory.StartNew(() => WriteLine("Ordering food."));
```

Since the task of invitation comes after ordering food, this time you'll notice the use of the `ContinueWith` method. This method creates a continuation that executes when the target task is completed. At the time of this writing, there are 20 overloads of this method. In this example, I use the simplest version of `ContinueWith` that accepts an `Action<Task>` as the parameter. This is why you will see the following code:

```
var task2 =
 task1.ContinueWith((t) => WriteLine("Inviting friends."));
```

Similarly, since arranging dinner comes after the second task (`task2`) finishes, you will see the following code:

```
var task3=
 task2.ContinueWith((t) => WriteLine("Arranging dinner."));
```

Finally, to show you the output messages, I'll wait for the final task (`task3`) to finish. This is why you will notice the following line at the end of the program:

```
task3.Wait();
```

## Demonstration 3

Here is the complete demonstration:

```
using static System.Console;

var task1 =
 Task.Factory.StartNew(() => WriteLine("Ordering food."));
var task2 =
 task1.ContinueWith((t) => WriteLine("Inviting friends."));
var task3=
 task2.ContinueWith((t) => WriteLine("Arranging dinner."));
task3.Wait();
```

## Output

Here is the output:

```
Ordering food.
Inviting friends.
Arranging dinner.
```

# Specialized Continuation

In the previous demonstration, `task3` followed `task2`, which in turn, followed `task1`. This flow guaranteed that `task3` could not be completed before `task1`. However, if `task2` runs independently and does not follow `task1`, there is no guarantee that `task3` will be completed after `task1` (it is because `task3` follows `task2` but not `task1`). So, if you want to confirm that `task3` should always continue only after the completion of other tasks, you need to have more control over the continuation process. Let's examine this type of specialized continuation with some case studies.

## Case Study 1

Let's assume you want to confirm that `task3` executes when both `task1` and `task2` complete their executions. In this case, you'll see me using the `ContinueWhenAll` method. As usual, there are many overloads of this method. I am about to use the following one that accepts two parameters. This method has the following form:

```
public Task ContinueWhenAll<TAntecedentResult>
(
  Task<TAntecedentResult>[] tasks,
  Action<Task<TAntecedentResult>[]> continuationAction
)
{
  // Method body not shown
}
```

Here, the first parameter accepts an array of antecedent tasks (which means that these need to be finished before you continue), and the next parameter is for the `Action` delegate that will execute when all tasks in the array have been completed.

This is why you'll see the following line that indicates that `task1` and `task2` must be completed before you start a continuation task:

```
var task3 = Task.Factory.ContinueWhenAll(
     new [] { task1, task2 },
     tasks =>
     {
         WriteLine("Arranging dinner.");
     }
);
```

## Demonstration 4

Let's see the complete program now:

```
var task1 = Task.Factory.StartNew(() => WriteLine("Ordering food."));
var task2 = Task.Factory.StartNew(() => WriteLine("Inviting
 friends."));

var task3 = Task.Factory.ContinueWhenAll(
     new [] { task1, task2 },
     tasks =>
     {
         WriteLine("Arranging dinner.");
     }
);

task3.Wait();
```

## Output

Here is some possible output where food is ordered at the beginning:

```
Ordering food.
Inviting friends.
Arranging dinner.
```

Here is some other possible output where the invitation is done at the beginning:

```
Inviting friends.
Ordering food.
Arranging dinner.
```

## Analysis

In every case, you can see that dinner has been arranged only after the task of ordering food is completed and the invitations are done.

**POINT TO REMEMBER**

You may note that the "Collection expressions" feature in C# 12 allows us to rewrite `task3` as follows (notice the change in bold for your reference):

```
// Using C#12's "Collection expression" feature
var task3 = Task.Factory.ContinueWhenAll(
     [task1, task2],
     tasks =>
     {
         WriteLine("Arranging dinner.");
     }
  );
```

## Q&A Session

**Q1.4 I can see that you did not use the lambda parameter `tasks` in the following code:**

```
tasks =>
{
    WriteLine("Arranging dinner.");
}
```

**This indicates that you could omit the parameter. Is this correct?**

Nice observation. I was forced to use this parameter to get support from the built-in construct. If you do not use the parameter in this place, you'll see the following error:

```
CS1593 Delegate 'Action<Task[]>' does not take 0 arguments
```

But this parameter is useful when you want to analyze the individual tasks that were finished before. For example, see the following program:

```
#region For Q&A session
var task1 = Task.Factory.StartNew(() => "Ordering food.");
var task2 = Task.Factory.StartNew(() => "Inviting friends.");
var task3 = Task.Factory.ContinueWhenAll(
     new Task<string>[] { task1, task2 },
     //new [] { task1, task2 }, // Ok too
     // [task1,task2], // C#12 onwards
     tasks =>
     {
         foreach (var task in tasks)
         {
             WriteLine(task.Result);
         }
         WriteLine("Arranging dinner.");
     }
     );
task3.Wait();
#endregion
```

This code can produce the following output:

```
Ordering food.
Inviting friends.
Arranging dinner.
```

**Author's Note:** The C# compiler can also infer the type `Task<string>`, as shown in the commented code. From C# 12 onward, you can further simplify the code, which is also shown in the commented line.

---

**Note** You can download the project `Demo4_Specialized Continuation` from the Apress website to test Demonstration 4 and the Q&A.

---

## Case Study 2

Let's analyze one more case study where you continue a task (say, `task3`) if any one of the previous tasks (say `task1` or `task2`) completes the execution. In this case, you can use `ContinueWhenAny` (instead of `ContinueWhenAll`). For example, you can replace `ContinueWhenAll` with `ContinueWhenAny` in the previous code, and you will get any of the following outputs:

**Possible Output 1:**

```
Ordering food.
Arranging dinner.
```

**Possible Output 2:**

```
Inviting friends.
Arranging dinner.
```

**Possible Output 3:**

```
Ordering food.
Inviting friends.
Arranging dinner.
```

**Possible Output 4:**

```
Inviting friends.
Ordering food.
Arranging dinner.
```

Nowadays computers are very fast. As a result, you may not see the possible case 1 or possible case 2. So, to demonstrate the working mechanism of `ContinueWhenAny`, let me simulate some delay inside `task1` and `task2`. I assume ordering food can be faster than inviting friends. So, I introduce `Thread.Sleep(10)` inside `task1` and `Thread.Sleep(100)` inside `task2`.

## Demonstration 5

Let's see the complete program now:

```
using static System.Console;

var task1 = Task.Factory.StartNew(
    () =>
    {
        Thread.Sleep(10);
        WriteLine("Ordering food.");
    }
    );
var task2 = Task.Factory.StartNew(
    () =>
    {
        Thread.Sleep(100);
        WriteLine("Inviting friends.");
    }
    );
```

```
var task3 = Task.Factory.ContinueWhenAny(
     new[] { task1, task2 },
    // [task1, task2], // C#12 onwards
     tasks =>
     {
         WriteLine("Arranging dinner.");
     }
     );

task3.Wait();
```

## Output

This time the probability of getting the following output is very high:

```
Ordering food.
Arranging dinner.
```

**Author's Note:** I hope that you now understand the concept of task continuations. I suggest you exercise a program where you check the result of an antecedent task before you continue a new task. I leave this as an exercise for you.

# Nested Tasks

The following code creates two `Task` instances, called `parent` and `child`. Notice that the `child` task is created inside the `parent` task.

```
using static System.Console;
var parent = Task.Factory.StartNew(
    () =>
    {
```

```
        WriteLine("The parent task has started.");
        var child = Task.Factory.StartNew(
            () =>
            {
                WriteLine("The child task has started.");
                // Forcing some delay
                Thread.Sleep(1000);
                WriteLine("The child task has finished.");
            });
        WriteLine("The parent task has finished now.");
    }
);
parent.Wait();
```

Here is some possible output once you run this program:

```
The parent task has started.
The parent task has finished now.
The child task has started.
```

Notice that this output does not reflect whether the `child` task is finished. Why? In the following documentation, Microsoft provides the following information (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/attached-and-detached-child-tasks):

> *A child task (or nested task) is a System.Threading.Tasks.Task instance that is created in the user delegate of another task, which is known as the parent task. A child task can be either detached or attached. A detached child task is a task that executes independently of its parent. An attached child task is a nested task that is created with the TaskCreationOptions. AttachedToParent option whose parent does not explicitly or by default prohibit it from being attached. A task may create any number of attached and detached child tasks, limited only by system resources.*

## Using TaskCreationOptions

The previous information is self-explanatory. You understand that in the previous code, the main thread waits for the parent task to finish, but it does not wait for the child task. To create the parent-child relationship, we need to attach the child task to the parent. Now the question is: how can we create such a relationship? See the following program that demonstrates how to use an overloaded version of the `StartNew` method to accept the following argument: `TaskCreationOptions.AttachedToParent`.

## Demonstration 6

Here is a sample program with the key changes in bold:

```
using static System.Console;
var parent =
    Task.Factory.StartNew(
    () =>
    {
        WriteLine("The parent task has started.");
        var child = Task.Factory.StartNew(
            () =>
            {
                WriteLine("The child task has started.");
                // Forcing some delay
                Thread.Sleep(1000);
                WriteLine("The child task has finished.");
            }, TaskCreationOptions.AttachedToParent);
        WriteLine("The parent task has finished now.");
    }
);
parent.Wait();
```

## Output

Run this program and notice the output. You can see that this time the child task completed its execution too:

```
The parent task has started.
The parent task has finished now.
The child task has started.
The child task has finished.
```

## Q&A Session

**Q1.5 In this demonstration, you have written this:**

```
parent.Wait();
```

**This means you are not waiting for the child task to finish. As a result, there is no guarantee that the output will reflect whether the child task has finished. Is this a correct understanding?**

No. Microsoft has designed the architecture in such a way that if you create a parent-child relationship, waiting on the parent task forces you to wait for the child task to complete.

**Q1.6 At the beginning of this section, when the child task was not attached to the parent, I saw that the output did not reflect whether the child task completed its execution. You explained the scenario saying that in that code segment, you waited only for the parent task but not for the child task. So, if I replace the line `parent.Wait();` with `Task.WaitAll(parent, child);`, I can see whether the child task finishes its execution. Am I right?**

No. In that code sample, the child task was nested. So, it was not in the scope. So, your proposed code will cause the following compile-time error:

```
CS0103 The name 'child' does not exist in the current context
```

## Using TaskContinuationOptions

You have already seen a demonstration where a parent task waits for a child task to finish. Also, I have printed a message in the console saying that the child task has finished. But there is a better way to check whether the task was completed properly. For example, if you want to verify the status of the child task, you can introduce another task that will continue after the child task as follows:

```
var statusChecker = child.ContinueWith(
 task =>
 {
  WriteLine($"Task id {task.Id}'s status is {task.Status}.");
 }, TaskContinuationOptions.AttachedToParent);
```

In this case, I use an overloaded version of the `ContinueWith` method where the first parameter accepts an `Action<Task>` and the next parameter indicates the **option for when the continuation is scheduled and how it behaves.** Figure 1-2 shows the various continuation options in Visual Studio.

```
namespace System.Threading.Tasks
{
    public enum TaskContinuationOptions
    {
        None = 0,
        PreferFairness = 1,
        LongRunning = 2,
        AttachedToParent = 4,
        DenyChildAttach = 8,
        HideScheduler = 16,
        LazyCancellation = 32,
        RunContinuationsAsynchronously = 64,
        NotOnRanToCompletion = 65536,
        NotOnFaulted = 131072,
        OnlyOnCanceled = 196608,
        NotOnCanceled = 262144,
        OnlyOnFaulted = 327680,
        OnlyOnRanToCompletion = 393216,
        ExecuteSynchronously = 524288
    }
}
```

***Figure 1-2.*** *The members of TaskContinuationOptions*

If you expand these options, you can see more detail in them. For example, I can expand `AttachedToParent` to get the information shown in Figure 1-3.

```
// Summary:
//     Specifies that the continuation, if it is a child task, is attached to a parent
//     in the task hierarchy. The continuation can be a child task only if its antecedent
//     is also a child task. By default, a child task (that is, an inner task created
//     by an outer task) executes independently of its parent. You can use the System.Threading
//     option so that the parent and child tasks are synchronized. Note that if a parent
//     task is configured with the System.Threading.Tasks.TaskCreationOptions.DenyChildAttach
//     option, the System.Threading.Tasks.TaskCreationOptions.AttachedToParent option
//     in the child task has no effect, and the child task will execute as a detached
//     child task. For more information, see Attached and Detached Child Tasks.
AttachedToParent = 4,
```

***Figure 1-3.*** *The detail of AttachedToParent in TaskContinuationOptions*

## Demonstration 7

Let's see an implementation. The following program is a modified version of the previous demonstration where we verify the status of the child task using the `Status` property. Notice the key changes in bold.

---

**Note** You can see that I have used `Task.CurrentId` to get the ID of the task that is currently executing. You can use the same for debugging purposes in a parallel environment.

---

```
using static System.Console;

var parent = Task.Factory.StartNew(
    () =>
    {
        WriteLine($"The parent task[id:{Task.CurrentId}] has
         started.");
        var child = Task.Factory.StartNew(
            () =>
            {
                WriteLine($"The child task[id:
                {Task.CurrentId}] has started.");
                Thread.Sleep(1000);
                WriteLine("The child task has finished.");
            }, TaskCreationOptions.AttachedToParent);

        var statusChecker = child.ContinueWith(
            task =>
            {
                WriteLine($"Task id {task.Id}'s status is
```

```
                {task.Status}.");
            },
            TaskContinuationOptions.AttachedToParent);

        WriteLine("The parent task has finished now.");
    }
);

parent.Wait();
```

## Output

Here is some sample output with the important changes in bold:

```
The parent task[id:9] has started.
The parent task has finished now.
The child task[id:10] has started.
The child task has finished.
Task id 10's status is RanToCompletion.
```

## Analysis

The life cycle of a `Task` instance passes through various stages. The status property is used to verify the current state. On investigation, you'll see that `TaskStatus` is an `enum` type and has many members. Figure 1-4 the members in Visual Studio.

```
namespace System.Threading.Tasks
{
    public enum TaskStatus
    {
        Created = 0,
        WaitingForActivation = 1,
        WaitingToRun = 2,
        Running = 3,
        WaitingForChildrenToComplete = 4,
        RanToCompletion = 5,
        Canceled = 6,
        Faulted = 7
    }
}
```

***Figure 1-4.*** *Different possible states of a Task instance*

In a concurrent environment, it is possible that by the time you receive the value of a task status, the status has changed. The interesting point is that once a state is reached, it does not go back to a previous state. For example, once a task reaches a final state, it cannot go back to the `Created` state. Interestingly, there are three possible final states as follows:

- `RanToCompletion`
- `Canceled`
- `Faulted`

As per their names, these states have their usual meaning. For example, `RanToCompletion` indicates that the task was completed successfully. Similarly, `Faulted` indicates the task completed due to an unhandled exception. I assume that I need not mention that `Canceled` indicates that a task was canceled, which can occur due to various reasons such as user intervention, timeouts, or any other application logic. I'll discuss cancellations and exceptions in the next chapter.

Now you understand that in the previous output, the task (which had ID 10) was completed successfully.

## Q&A Session

**Q1.7 In the previous output, I see the IDs of the parent task and the child task are 9 and 10, respectively. It looks like many other tasks were also running. Is this correct?**

Yes, I ran this code in Visual Studio 2022 with the default settings in the debug configuration where the hot reload was enabled. I asked the same question at https://stackoverflow.com/questions/77726578/vs2022-versus-vs2019-how-why-are-the-additional-tasks-being-created and received the answer. If you run the same code in the release configuration (or disable the "hot reload" setting), you can see the following output:

```
The parent task[id:1] has started.
The parent task has finished now.
The child task[id:2] has started.
The child task has finished.
Task id 2's status is RanToCompletion.
```

**Author's Note:** To choose your preferred configuration, you can follow these steps: **right-click the Solution Explorer, select Configuration Properties and Configuration, and then choose Debug or Release configuration.**

**POINT TO REMEMBER**

I often execute my programs in debug mode. So, to get the lower task IDs such as 1, 2, 3, and so forth in the output, I often run those programs with the "hot reload" setting disabled.

**Q1.8 In your example, the child task was completed successfully. But you told me that the final state of a task can be different. For example, a child task can encounter an exception or someone can cancel it while it is running. How can you handle these scenarios?**

You can use separate branches to handle different scenarios. Since I have not discussed task cancellation and exception handling yet, I showed you an example that could be completed easily. Once you learn about handling those special scenarios in Chapter 2, the following program will be easy for you to understand. For now, you can concentrate on the following code that demonstrates two possible cases where a child task can finish successfully if a user does not press C immediately (more specifically, within five seconds once the child task starts). Otherwise, it will be canceled. This is why you'll see the following two branches:

```
var successHandler = child.ContinueWith(
    task =>
    {
        WriteLine($"Task id: {task.Id} finished successfully");
    },
    TaskContinuationOptions.AttachedToParent |
    TaskContinuationOptions.OnlyOnRanToCompletion);

var cancelHandler = child.ContinueWith(
   task =>
   {
       WriteLine($"Task id:{task.Id} was canceled.");
   },
   TaskContinuationOptions.AttachedToParent |
   TaskContinuationOptions.OnlyOnCanceled);
```

## Demonstration 8

I have included this program to answer your question only. Since I have not discussed cancellations yet, you may not understand all the lines in this program. For now, you can focus on the two branches where one branch handles the successful completion of a child task and the other branch handles the cancellation of the child task. Still, if you find it difficult to understand, you can come back here after you learn how to cancel a task (discussed in Chapter 2).

Here is the complete program:

```
using static System.Console;

var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

var parent = Task.Factory.StartNew(
    () =>
    {
        WriteLine("The parent task has started.");
        var child = Task.Factory.StartNew(
            () =>
            {
                WriteLine($"The child task[id:
                 {Task.CurrentId}] has started.");
                Thread.Sleep(5000);
                token.ThrowIfCancellationRequested();
                WriteLine("The child task has finished.");
            },
            token,
            TaskCreationOptions.AttachedToParent,
            TaskScheduler.Default
     );
```

```
        var successHandler = child.ContinueWith(
            task =>
            {
                WriteLine($"Task id: {task.Id} finished
                successfully.");
            },
            TaskContinuationOptions.AttachedToParent |
            TaskContinuationOptions.OnlyOnRanToCompletion);

        var cancelHandler = child.ContinueWith(
           task =>
           {
               WriteLine($"Task id: {task.Id} was canceled.");
           },
           TaskContinuationOptions.AttachedToParent |
           TaskContinuationOptions.OnlyOnCanceled);

        WriteLine("The parent task has finished now.");
    }
);

WriteLine("Press 'c' immediately to cancel the child task.");
char ch = ReadKey().KeyChar;
if (ch == 'c')
{
    WriteLine("\nTask cancellation requested.");
    tokenSource.Cancel();
}
parent.Wait();
```

## Output

Here is the sample output when a user does not cancel the task:

```
Press 'c' immediately to cancel the child task.
The parent task has started.
The parent task has finished now.
The child task[id:2] has started.
The child task has finished.
Task id: 2 finished successfully.
```

Here is another sample output when the user raised a request to cancel the child task:

```
Press 'c' immediately to cancel the child task.
The parent task has started.
The parent task has finished now.
The child task[id:2] has started.
c
Task cancellation requested.
Task id: 2 was canceled.
```

# Discussion of Waiting

When you execute a task and analyze the outcome, you must wait for the task to finish. Now the question is: how can you wait? There are different built-in constructs for this, and by using any of them, you can design your application. In this chapter, you have already seen me using some of those constructs.

In this section, I will show you the need to "wait," and I will discuss some useful methods to implement this idea.

**Note** Waiting for a task literally blocks the current thread. So, in a parallel execution environment, you must be careful before implementing the idea.

## Why Do We Wait?

First, let's understand why you need a waiting mechanism. The following demonstration will give you an idea.

### Demonstration 9

Here is a program that runs two simple tasks. Let's execute the following program and analyze some of the possible outputs:

```
using static System.Console;

WriteLine("The main thread starts.");

var task1 = Task.Run(
    () =>
    {
        WriteLine("Task1 starts.");
        WriteLine("Task1 ends.");
    }
);

var task2 = Task.Run(
    () =>
    {
        WriteLine("Task2 starts.");
```

```
        WriteLine("Task2 ends.");
    }
);
WriteLine("The end of main.");
```

## Output

Here are two possible outputs.

**Output 1:**

```
The main thread starts.
The end of main.
```

**Output 2:**

```
The main thread starts.
The end of main.
Task1 starts.
```

## Analysis

These outputs reflect the following characteristics:

- The main thread ends before `task1` and `task2` finish their executions.
- None of these outputs reflects whether `task1` or `task2` completes their job.

# How Do We Wait?

To see the final status of these tasks, you may need to wait. How can you do that? There are different approaches. Let me show you some of them in the following section.

## Using Sleep

One of the simplest solutions is to block the main thread until the other tasks are finished. Here is a sample where I block the main thread for 1,000 milliseconds:

```
// The previous code is the same
Thread.Sleep(1000);
WriteLine("The end of main.");
```

This one line of additional code increases the probability of seeing some output that reflects that `task1` and `task2` complete their executions before the control leaves the main thread. Here is some possible output:

```
The main method starts.
Task2 starts.
Task1 starts.
Task1 ends.
Task2 ends.
The end of main.
```

The advantage of using this approach is obvious. We can see that when the main thread sleeps, the other tasks could execute their jobs. This indicates that during sleep, the scheduler can schedule other tasks.

On the contrary, this approach has an obvious problem: you may block the thread unnecessarily for some additional time. For example, I can see the same output on my computer if I block the main thread for 500 milliseconds or less. **However, the problem is that since we cannot predict the exact time for these tasks to be completed, I need to block it for a reasonable amount of time.** Still, if any of these tasks take more time to complete due to some other factors, you may not see any of the task's completion messages in the final output. This is a problem for sure!

We do not want unnecessary waiting. At the same time, we do not want to miss any key information. From this point of view, this is an inefficient approach. In fact, the situation can be worse if you work on an application that tries to block the UI. This is why relying on the `Sleep` method may not always be a good idea.

## Using Delay

If you replace the `Thread.Sleep` methods with `Task.Delay` methods, you may see similar output. For example, in the previous code, let me replace the statement `Thread.Sleep(1000);` with `Task.Delay(1000);` and run the program again. Again, my computer shows different possible outputs, and one of them is as follows:

```
The main thread starts.
Task1 starts.
Task1 ends.
The end of main.
Task2 starts.
Task2 ends.
```

This output reflects that `task1` and `task2` completed their jobs. However, notice that the calling thread was not blocked this time. So, you see the line `The end of main.` before the line `Task2 starts.`

In fact, Visual Studio says that when you use the `Delay` method, the call is not awaited and the calling thread can continue before the call is completed (see the message shown in Figure 1-5).

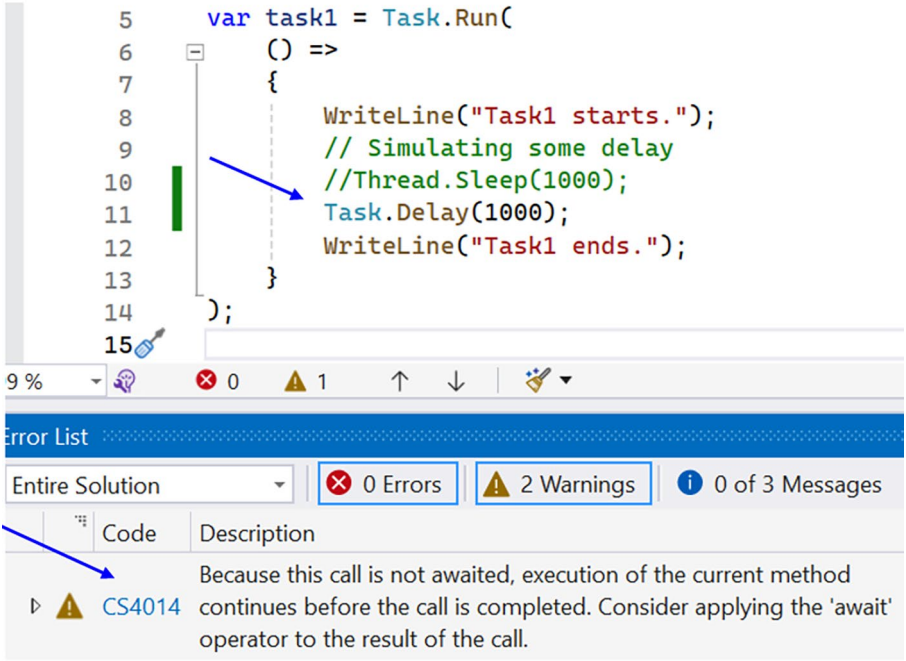



***Figure 1-5.*** *Visual Studio confirms that the call is not awaited*

This gives you a clue that you should use `Sleep` for the synchronous pauses whereas you should prefer the `Delay` method for nonblocking delays. In this book, you'll also see me using the `Delay` method in many demonstrations (particularly, in Chapter 6 when I'll use the `async` and `await` keywords in my programs). The reason is that `Task.Delay` is more suitable in asynchronous programming. While answering the question Q1.12 in the Q&A Session, I'll compare the `Sleep` method and the `Delay` method in more detail.

---

**Note** Using the `async` and `await` keywords simplifies asynchronous programming. They are explained in Chapter 6. You do not need to worry about them now.

---

## Using ReadKey()

Sometimes you will see the presence of `ReadKey()`, `Read()`, or `ReadLine()` in a program. The basic idea of using these methods is to block control of execution until the user provides the required input. For example, you can wait for `task1` and `task2` to finish, and then you can press a key from the keyboard to get the final output. Here is some sample code:

```
// The previous code is the same
ReadKey();
WriteLine("The end of main");
```

## Using Wait

Earlier you saw that by using the `Result` property, I blocked the calling thread until the specified tasks were finished. This means I was waiting for those tasks to be completed. It is not necessary in every scenario to analyze the outcome of task execution. In fact, a task may not return a value at all. So, let's search for alternatives for the waiting techniques.

First, let me remind you about the `Wait` method that you have seen already. When invoking the `Wait` method on a `Task` instance, you can wait for it to complete. Here is a sample where I call `Wait` on `task1` and `task2` separately:

```
// The previous code is the same
task1.Wait();
task2.Wait();
WriteLine("The end of main.");
```

Here is some sample output after this change:

```
The main thread starts.
Task2 starts.
Task2 ends.
```

```
Task1 starts.
Task1 ends.
The end of main.
```

## Using WaitAll

Instead of waiting for the individual tasks to be completed, you can wait for a group of tasks. In such cases, you use the `WaitAll` method and provide the task objects for which you want to wait as parameters. Here is a sample:

```
// The previous code is the same
Task.WaitAll(task1, task2);
WriteLine("The end of main.");
```

This change can also produce an output that reflects both tasks finished their execution.

---

**Note** You can download the project `Demo9_DiscussionOnWaiting` from the Apress website to run and validate all these program segments on your computer.

---

## Using WaitAny

Suppose there are multiple tasks, but you'd like to wait for any of them to complete. In such cases, you use the `WaitAny` method as follows:

```
// The previous code is the same
Task.WaitAny(task1, task2);
WriteLine("The end of main.");
```

To show you the effect of this change, let me introduce a small delay in `task2` as follows:

```
var task2 = Task.Run(
    () =>
    {
        WriteLine("Task2 starts.");
        // A small delay is introduced here
        Thread.Sleep(10);
        WriteLine("Task2 ends.");
    }
);
```

Run the program now. From these outputs, you will see that the main thread completes when at least one of these tasks finishes its execution. Here I have included some samples that show by the time the main thread finished, `task1` completed its execution, but `task2` could not do that.

**Output-1:**

```
The main thread starts.
Task1 starts.
Task1 ends.
The end of main.
Task2 starts.
```

**Output-2:**

```
The main thread starts.
Task1 starts.
Task1 ends.
The end of main.
```

**POINTS TO NOTE**

Remember the following points:

- You may see different output on your computer.
- These methods have various overloads. For example, at the time of this writing, the `Wait` method has six different overloads. Using these overloaded versions, you can provide a maximum duration to wait, a `CancellationToken` instance, or both of them to monitor while waiting.

## Using WhenAny

Notice the previous outputs once again. You can see that the line `The end of main.` came after `task1` finished its execution. If you execute the program repeatedly, you'll never see that the mentioned line appears before any of these tasks finish the execution. That is because, **in the case of** `WaitAny`**, the calling thread is blocked until any of those tasks finishes the execution.** Interestingly, there is another method, called `WhenAny`, that does not block the calling thread.

Consider the following code where I replace `WaitAny` with `WhenAny`:

```
// The previous code is the same
Task.WhenAny(task1, task2);
WriteLine("The end of main.");
```

Here is some sample output where you can see that the main thread completes before `task1` and `task2`:

```
The main thread starts.
Task1 starts.
The end of main.
Task1 ends.
Task2 starts.
```

To make the concept clearer, let me apply some modifications to the existing program as follows:

```
// The previous code is the same
var task=Task.WhenAny(task1, task2);
WriteLine("The end of main.");
WriteLine($"Task1's ID: {task1.Id} Task2's ID: {task2.Id}");
WriteLine($"Completed task id: {task.Result.Id}");
```

Run this modified program now. The following output confirms that the use of `WhenAny` does not block the calling thread. This is why I include a possible output to show you that the line `The end of main.` appeared before `task1` or `task2` finished their executions. Here is some sample output:

```
The main thread starts.
The end of main.
Task1's ID: 1 Task2's ID: 2
Task1 starts.
Task1 ends.
Completed task ID: 1
Task2 starts.
```

---

**Note** You can download the project `Demo10_DiscussingWaitAnyAndWhenAny` from the Apress website to run and validate the new program segments on your computer.

---

## Waiting for Cancellation

There will be situations when you need to be prepared for possible cancellations of tasks. In those cases, you need to have a cancellation token. Since the topic is important as well as big, I will discuss it in the next chapter.

## Q&A Session

**Q1.9 In some articles, I see the usage of `Thread.SpinWait` instead of `Thread.Sleep`. How do they differ?**

The `SpinWait` method is useful for implementing locks but not for ordinary applications. When you use spin waiting, the scheduler does not pass the control to some other task, which means it avoids context switching. The documentation at https://learn.microsoft.com/en-us/dotnet/api/system.threading.thread.spinwait?view=net-7.0 states the following:

> *In the rare case where it is advantageous to avoid a context switch, such as when you know that a state change is imminent, make a call to the SpinWait method in your loop. The code SpinWait executes is designed to prevent problems that can occur on computers with multiple processors. For example, on computers with multiple Intel processors employing Hyper-Threading technology, SpinWait prevents processor starvation in certain situations.*

---

**Note** .NET Framework classes such as `Monitor` and `ReaderWriterLock` internally use the `SpinWait` method. Still, instead of using this method directly, Microsoft recommends that you use the built-in synchronization classes to serve your purpose. (I discuss different synchronization techniques in Chapter 3 and Appendix A.) I also recommend not using this method for one typical reason: `SpinWait` accepts an integer argument that represents the number of iterations for the CPU loop to be performed. As a result, the waiting time depends on the processor's speed.

---

It is useful to note that you can use the `SpinUntil` method as well. At the time of this writing, there are three overloaded versions of this method.

```
SpinUntil(Func<Boolean>)
SpinUntil(Func<Boolean>, Int32)
SpinUntil(Func<Boolean>, TimeSpan)
```

Let me show you how to use the simplest version that spins until the specified condition is fulfilled. Here is an example where I wait until `task1` completes its execution properly:

```
// Previous code is the same.
SpinWait.SpinUntil(() => task1.Status == TaskStatus.RanToCompletion);
WriteLine("The end of main.");
```

Here is some possible output after this change is made to this program:

```
The main thread starts.
Task2 starts.
Task1 starts.
Task1 ends.
The end of main.
```

Note that this time the output shows that `task1` completes the execution, but it does not reflect whether `task2` completes the execution. So, the key takeaway is that there are different ways of waiting. You can use the one that is more convenient for you.

**POINT TO NOTE**

I am mentioning only those methods that will be sufficient for you to understand the rest of this book. As said before, these methods have various overloaded versions as well.

**Q1.10 "These methods have various overloaded versions as well." How can I know about them?**

You can follow any of the traditional approaches. For example, you can use Visual Studio's intelligence or read Microsoft's documentation to learn about them. For example, Figure 1-6 shows that the `Sleep` method has two overloaded versions and the first one accepts an `int` parameter.

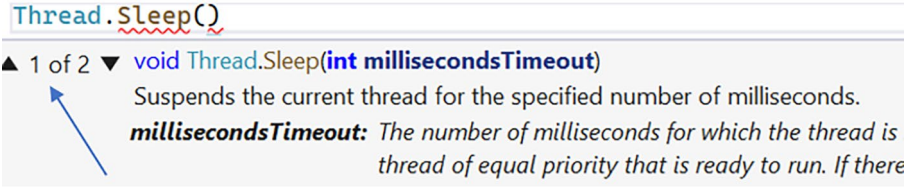

Thread.Sleep()
▲ 1 of 2 ▼ void Thread.Sleep(int millisecondsTimeout)
Suspends the current thread for the specified number of milliseconds.
millisecondsTimeout: The number of milliseconds for which the thread is
thread of equal priority that is ready to run. If there

***Figure 1-6.*** *The Sleep method has two overloaded versions*

**Q1.11 Can you give me an example where you'd use `WhenAny` or `WaitAny`. Between these two methods, which one would you like to use?**

Suppose you are working with two different tasks and each task works with different URLs. Let's further assume that each URL can help you test the current health of a website. You understand that any of these links will be sufficient to check the current status of a website. So, your program can execute the tasks and continue as soon as you get the data. In such a case, you can use `WhenAny` or `WaitAny`.

Unless there are sufficient reasons, I'll opt for `WhenAny` in such a case, because it is nonblocking. This is because if a task waits to get a notification from a set of tasks where none can be completed due to some unpredictable circumstances, the use of `WaitAny` can cause a deadlock as well.

**Q1.12 Between `Thread.Sleep` and `Task.Delay`, which one do you prefer?**

The `Sleep` method suspends the calling thread, which may not be ideal in every scenario. The problem was already discussed, so I won't discuss it here. However, for a simple demonstration, this method is handy; so, I used it in other demonstrations as well. Once you complete this book, I assume that you'll have a fair idea of how asynchronous programming works in a parallel execution environment. In most cases, you'll want to use the `Delay` method. The obvious reason for this is that invoking the `Delay` method does not cause the calling thread to be blocked. This is why the use of the `Delay` method can help you build a more responsive UI.

Once you're familiar with `async/await` programming, you'll learn that you can write something like the following: `await Task.Delay(1000);` however, you cannot write something like `await Task.Sleep(10);`.

While using the `Delay` method, you can also assign it to a task and await at a later point in time as follows:

```
Task task= Task.Delay(1000);
// Do something here
await task;
```

You should also note that the `Delay` method has various overloads, and many of them accept `CancellationToken` as a parameter (this is discussed in the next chapter). Using this parameter, you can avoid aborting the thread and terminate it nicely.

# Exercises

Check your understanding by attempting the following exercises.

**POINT TO REMEMBER**

As mentioned, for all code examples, the "Implicit Global Usings" was enabled in Visual Studio. This is why you won't see me mentioning the following namespaces that are available by default:

```
System
System.Collections.Generic
System.IO
System.Linq
System.Net.Http
System.Threading
System.Threading.Tasks
```

The same comment applies to all the exercises in this book.

**E1.1** Can you predict the output of the following program?

```
using static System.Console;

Task printHelloTask = new (
    () => WriteLine("Hello!")
);
WriteLine("End.");
```

**E1.2** Can you predict the output of the following program?

```
using static System.Console;

Task printHelloTask = new(
    () => WriteLine("Hello!")
);
printHelloTask.Start();
WriteLine("End.");
```

**E1.3** Can you predict the output of the following program?

```
using static System.Console;
var saySomething = (string msg = "Hello") => msg;

var printHelloTask = Task.Factory.StartNew(
    () =>
    {
        Thread.Sleep(1000);
        WriteLine(saySomething());
    }
);
printHelloTask.Wait();
WriteLine("End.");
```

**E1.4** Can you predict the output of the following program?

```
using static System.Console;

var calculate = Task.Run(() => GetTotal(5));
WriteLine(calculate.Result);
WriteLine("End.");

static int GetTotal(int count)
{
    int total = 0;
```

```
    for (int i = 1; i <= count; i++)
    {
        total += i*2;
    }
    return total;
}
```

**E1.5** Starting with C# 12, we can define a primary constructor as a part of the class declaration. Here is an example:

```
class Employee( string name,int id)
{
    private string _name=name;
    private int _id = id;
    public override string ToString()
    {
        return $"Name:{_name} Id:{_id}";
    }
}
```

You can create an instance of the `Employee` class as follows:

```
Employee emp = new("Bob", 1);
```

Now assume that there are two tasks where the first task will create an `Employee` instance. The second task will follow the first task and perform the following steps: first, it will verify whether the first task completes the process successfully. In addition, it will print the current date and time. Can you write a program fulfilling the criteria? (You do not need to handle exceptions or cancellations for this exercise.)

**E1.6** Consider two websites such as `www.google.com` and `www.yahoo.com`. You may know that a hostname can be associated with multiple IP addresses. Can you write a program that involves multiple tasks where one task prints the IP addresses associated with the hostnames and another

task pings these websites and displays the result? (Assume that your computer is already connected to the Internet. In addition, you do not need to handle exceptions or cancellations for this exercise.)

**E1.7** Can you predict the output of the following program?

```
using static System.Console;

var helloTask = Task.Run(() =>
{
    WriteLine("Hello reader!");
    var aboutTask = Task.Factory.StartNew(() =>
    {
        Task.Delay(1000);
        WriteLine("How are you?");
    }, TaskCreationOptions.AttachedToParent);
});

helloTask.Wait();
```

**E1.8** State **True**/**False:**

i) There are three possible final states of a task: `RanToCompletion`, `Canceled`, and `Faulted`.

ii) When you create a child task inside a parent task, it is detached by default.

iii) A parent task always waits for its child task to finish.

iv) Waiting for a task completion blocks the calling thread.

v) The `WaitAny` method blocks the calling thread, but the `WhenAny` method does not block the calling thread.

# Summary

This chapter gave you a quick overview of task programming. The discussion started by introducing you to the TPL and described various aspects of task creation and execution. It also described different approaches to implementing a waiting mechanism for task completion.

In brief, it answered the following questions:

- What is a task, and how can you create a task?
- How can you pass values into tasks and return a value from a task?
- How can you implement simple task continuation as well as a conditional continuation mechanism?
- How can you create branches to employ a conditional continuation mechanism?
- How can you check the status of the current task?
- How can you employ a waiting mechanism to ensure a task finishes its execution before the application ends?

# Solutions to Exercises

Here is a sample solution set for the exercises in this chapter.

## E1.1

The program will output the following:

```
End.
```

Notice that you have created the task but you have not started this task.

# E1.2

The program can show more than one possible output. You may think that it will print the following:

```
Hello!
End.
```

But, notice that you did not wait for the task to finish its execution. So, you may see `End.` before you see `Hello`.

```
End.
Hello!
```

It is also possible that in the output, you see `End.` only if the task takes some extra time to start. To examine this, you can run the following code in which by introducing some delay inside the task, I increase the probability of finishing the main thread early:

```
using static System.Console;

Task printHelloTask = new(
    () =>
    {
        Thread.Sleep(1000);
        WriteLine("Hello!");

    }
);
printHelloTask.Start();
WriteLine("End.");
```

## E1.3

C# 12 allows you to define default values for parameters on lambda expressions. So, the code will compile without any issues. As a result, this time the output is also predictable because the main thread must wait for the task to finish. So, you will see the following output:

```
Hello!
End.
```

## E1.4

You will see the following output:

```
30
End.
[Clue: 1*2+2*2+3*2+4*2+5*2=30]
```

## E1.5

Here is a sample program based on the features that you learned in this chapter:

```
using static System.Console;

var createEmp = Task.Factory.StartNew(
    () =>
        {
            Employee emp = new("Bob", 1);
            WriteLine($"Created an employee with {emp}");
        }
    )
```

```
    .ContinueWith(
      task =>
      {
          WriteLine($"Was the previous task completed?
           {task.IsCompletedSuccessfully}");
          WriteLine($"Current time:{DateTime.Now}");
      }
    );
createEmp.Wait();

class Employee( string name,int id)
{
    private string _name=name;
    private int _id = id;
    public override string ToString()
    {
        return $"Name: {_name} Id: {_id}";
    }
}
```

Here is some sample output:

```
Created an employee with Name: Bob Id: 1
Was the previous task completed? True
Current time:12/24/2023 12:06:25 PM
```

# E1.6

Here is a sample program that fulfills the criteria:

```
using System.Net;
using System.Net.NetworkInformation;
using static System.Console;

List<string> websites = ["www.google.com", "www.yahoo.com"];
```

```
// Same as:
// List<string> websites = new() {
// "www.google.com", "www.yahoo.com" };

var task1= Task.Run(()=>PrintHostNameAndIPs(websites));
var task2 = Task.Run(() => PingAll(websites));

Task.WaitAll(task1, task2);

void PrintHostNameAndIPs(List<string> urls)
{
    urls.ForEach(url =>
    {
        IPAddress[] ips = Dns.GetHostAddresses(url);
        WriteLine($"{url}'s IP addresses are:");
        ips.ToList().ForEach(WriteLine);
        WriteLine("--------------");

    });
}

void PingAll(List<string> urls)
{
    urls
    .Select(PingSite)
    .ToList()
    .ForEach(url => WriteLine($"{url.Address} ping
     status: {url.Status}  "));
}

static PingReply PingSite(string url)
{
    return new Ping().Send(url);
}
```

Here is some sample output:

```
www.google.com's IP addresses are:
2404:6800:4009:826::2004
142.250.183.196
--------------
www.yahoo.com's IP addresses are:
2406:8600:f040:1fa::3000
2406:8600:f040:1fa::2000
27.123.43.204
27.123.43.205
--------------
2404:6800:4009:826::2004 ping status: Success
2406:8600:f040:1fa::3000 ping status: Success
```

**POINT TO REMEMBER**

I have used the "Collection expressions" feature that was introduced in C# 12. This feature allows you to create a list as follows:

```
List<string> websites = ["www.google.com", "www.yahoo.com"];
```

This line of code is equivalent to the following:

```
List<string> websites = new()
{
  "www.google.com",
  "www.yahoo.com"
};
```

## Additional Note

The shown program looks more functional and less object-oriented. If you are biased toward OOP, you can consider the following alternative solution with the key changes in bold:

```
using System.Net;
using System.Net.NetworkInformation;
using static System.Console;

List<string> websites = ["www.google.com", "www.yahoo.com"];

var task1 = Task.Run(() => PrintHostNameAndIPsOOP(websites));
var task2 = Task.Run(() => PingAllOOP(websites));

Task.WaitAll(task1, task2);

void PrintHostNameAndIPsOOP(List<string> urls)
{
    foreach (string url in urls)
    {
        IPAddress[] ips = Dns.GetHostAddresses(url);
        WriteLine($"{url}'s IP addresses are:");
        foreach (IPAddress ip in ips)
        {
            WriteLine(ip);
        }
        WriteLine("--------------");

    }
}

void PingAllOOP(List<string> urls)
{

    List<PingReply> pingReplies = new();
```

```
    foreach (string url in urls)
    {
        pingReplies.Add(PingSite(url));
    }
    foreach (PingReply reply in pingReplies)
    {
        WriteLine($"{reply.Address} ping status:
         {reply.Status}");
    }
}

static PingReply PingSite(string url)
{
    return new Ping().Send(url);
}
```

# E1.7

On my computer, this program shows the following output:

```
Hello reader!
```

This is because the application terminated before `aboutTask` finishes its execution. Using the `ReadKey` or `ReadLine()` method you can hold the control until you see the following output:

```
Hello reader!
How are you?
```

You may wonder about this. Notice that I have used the `Run` method, but not the `StartNew` method. Implement the following change (shown in bold):

```
var helloTask = Task.Factory.StartNew(() =>
{
    WriteLine("Hello reader!");
    var aboutTask = Task.Factory.StartNew(() =>
    {
        Task.Delay(1000);
        WriteLine("How are you?");
    }, TaskCreationOptions.AttachedToParent);
});
helloTask.Wait();
```

You can expect to see the following output:

```
Hello reader!
How are you?
```

In this context, I also suggest you see the online documentation that states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.startnew?view=net-8.0):

> *Starting with the .NET Framework 4.5, you can use the Task.Run(Action) method as a quick way to call StartNew(Action) with default parameters. Note, however, that there is a difference in behavior between the two methods regarding : Task.Run(Action) by default does not allow child tasks started with the TaskCreationOptions.AttachedToParent option to attach to the current Task instance, whereas StartNew(Action) does.*

# E1.8

The answers are as follows:

i) There are three possible final states of a task- `RanToCompletion`, `Canceled`, and `Faulted`. [**True**]

ii) When you create a child task inside a parent task, it is detached by default.[**True**]

iii) A parent task always waits for its child task to finish. [**False**]

[**Clue**: In the case of an attached child task, the statement is true; but for a detached child task, it is false.]

iv) Waiting for a task completion blocks the calling thread. [**True**]

v) The `WaitAny` method blocks the calling thread, but the `WhenAny` method does not block the calling thread. [**True**]

CHAPTER 2

# Handling Special Scenarios

In Chapter 1, you learned about various aspects of tasks including continuation tasks and nested tasks. In this chapter, I'll continue the discussion of task programming, but this time the focus will be handling special scenarios such as exceptions and cancellations. Let's get started.

## Introduction to Exceptions

In a multithreaded environment, handling exceptions can be tricky. The reason is obvious: different tasks may throw different exceptions. Since you are reading about the advanced concepts of programming, I assume you are familiar with the fundamentals of exceptions and how to handle them in a C# project, so I will not discuss the basics in this book. Instead, I'll focus on possible exceptional scenarios when you program with tasks in a multithreaded environment.

## Understanding the Challenge

Let's start with a simple class, called `Product`. This class has a method, called `CheckUser`, to identify whether a user is a valid user. For simplicity,

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5_2

let's assume the following characteristic: **Each valid user ID starts with u. Otherwise, the `CheckUser` method throws an instance of `UnauthorizedAccessException`.**

## Demonstration 1

Now let me hack the system to create this exception and handle it inside a `try-catch` block. Here is a sample demonstration:

```
using static System.Console;

WriteLine("Exception handling demo.");

try
{
    var validateUser = Product.CheckUser("abc");
    WriteLine(validateUser);
}

catch (Exception e)
{
    WriteLine($"Caught error: {e.Message}");
}

class Product
{
    /// <summary>
    /// This method throws an exception when the user ID does
    /// not start with 'u'
    /// </summary>
    /// <param name="userId"> The user ID</param>
    /// <returns>Confirming the valid user</returns>
    /// <exception cref="UnauthorizedAccessException"> The
    /// exception is thrown when the <paramref
    /// name="userId" is an invalid ID </exception>
```

```
    public static string CheckUser(string userId)
    {
        string msg;
        if (userId.StartsWith("u"))
        {
            msg = $"{userId} is a valid user";
        }
        else
        {
            throw new UnauthorizedAccessException($"Id:
             {userId} is invalid.");

        }
        return msg;
    }
}
```

## Output

There is no surprise that upon executing this program, you'll see the following output:

```
Exception handling demo.
Caught error: Id: abc is invalid.
```

You are probably familiar with this type of program and there is nothing new up to this point.

**POINT TO NOTE**

Visual Studio can still display this exception if you want. For example, in Visual Studio's Exception Settings (Ctrl+Alt+E), if you want your program to break when a particular exception is thrown, you can enable the checkbox for that

particular exception. For example, I have the settings shown in Figure 2-1 for the common language runtime exceptions.

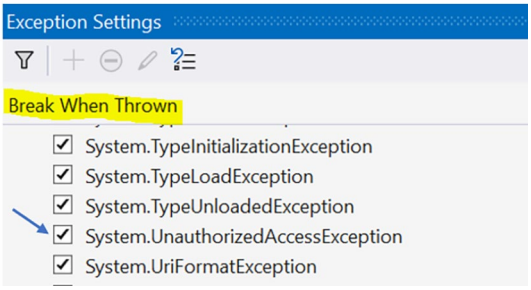



***Figure 2-1.*** *Exception Settings for Common Language Runtime exceptions in Visual Studio*

While executing the program, you will see the exception shown in Figure 2-2.

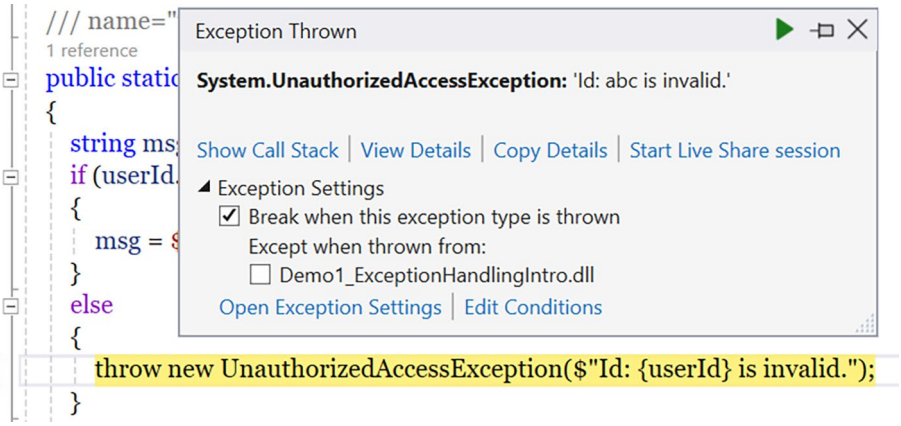



***Figure 2-2.*** *Visual Studio displays the UnauthorizedAccessException while the program was running*

Note that the program has not crashed yet; you can click the Continue button to resume the execution.

## Demonstration 2

Now change the program a little bit and analyze the situation again. This time, let's invoke the `CheckUser` method through a task that is created inside the main thread, as shown in the following program:

```
using static System.Console;

WriteLine("Exception handling demo.");

try
{
  var validateUser = Task.Run(() => Product.CheckUser("abc"));
  WriteLine("End");
}

catch (Exception e)
{
    WriteLine($"Caught error: {e.Message}");
}

// The Product class is the same. It is not shown to avoid
// repetition.
```

## Output

Upon executing this program, you may see the following output:

```
Exception handling demo.
End
```

You can see that this time the output does not show anything about the exception. **Why does this happen?** Notice that this time the main thread did not encounter the exception; it was encountered by the task called `validateUser` that was created by this main thread. However, an unobserved exception can cause problems at a later stage. So, you may be

interested in watching all the exceptions and decide to handle them as per the priority. **This is an important note for you to remember.**

**POINT TO NOTE**

I'd like to remind you that many developers like to use `Task.Run` instead of `Task.Factory.StartNew`. I also belong to that category. So, from this time onward, I'll be using `Task.Run` much more compared to `Task.Factory.StartNew`. In this context, you may want to revisit Q&A 1.2 in Chapter 1. In the end, it's up to you, so choose the approach that you prefer. Once you finish Chapter 6 of this book, you can read an online post comparing these approaches at https://devblogs.microsoft.com/pfxteam/task-run-vs-task-factory-startnew/. Though the article was written a while ago, but it is still useful.

## Retrieving the Error Details

How can I show you information about the exception? An obvious way is to handle the exception inside the task itself. For example, I can replace the following line:

```
var validateUser = Task.Run(() => Product.CheckUser("abc"));
```

with something like the following:

```
var validateUser = Task.Run(
    () =>
    {
        string msg = string.Empty;
        try { msg = Product.CheckUser("abc"); }
        catch (Exception e)
        {
            WriteLine($"Caught error inside the task:
```

```
                {e.Message}");
            }
            return msg;
        });
```

However, you understand that inside the main thread, we can launch many tasks, and if we code like this, there will be a lot of code duplication.

So, let's search for alternative approaches. Interestingly, in the main thread, if you use `WriteLine(validateUser.Result);` or `validateUser.Wait();` inside a `try` block, you can observe the exceptions. Let's see the next program.

## Demonstration 3

Here is a sample demonstration where I use the statement `validateUser.Wait();` inside the `try` block as follows:

```
// Previous code as it is

try
{
  //Product.CheckUser("abc");
  var validateUser = Task.Run(() => Product.CheckUser("abc"));
  validateUser.Wait();
  WriteLine("End");
}
// There is no change in the remaining code as well
```

## Output

Once you execute the program again, you will see the following output:

```
Exception handling demo.
Caught error: One or more errors occurred. ('abc' is an
 invalid user.)
```

The output depicts that `abc` is an invalid user and it was the key information that I wanted to display. If you put a breakpoint inside the `catch` block as shown in Figure 2-3 and debug this code, you can see the values shown.

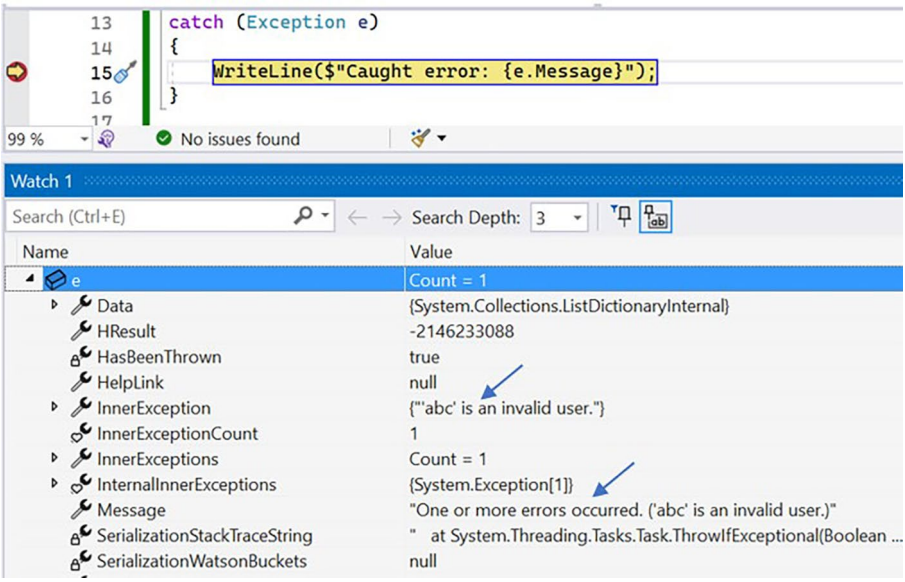



***Figure 2-3.*** *A partial snapshot from Visual Studio that shows an InnnerException*

You can see that "`'abc' is an invalid user.`" is an `InnerException`. For a situation like this, we can simply pass through the inner exceptions and display the error detail.

## Q&A Session

**Q2.1 In the previous output I can see that "`'abc' is an invalid user.`" was wrapped as an `InnerException`. Is there any specific reason for this?**

Good question. It is because the program caught the `AggregateException` exception. To see this, you can slightly modify the `catch` block as follows:

```
catch (Exception e)
{
    WriteLine($"Caught error: {e.Message}");
    WriteLine($"Exception name: {e.GetType().Name}");
}
```

If you execute the application again, you will see the following output:

```
Exception handling demo.
Caught error: One or more errors occurred. ('abc' is an
 invalid user.)
Exception name: AggregateException
```

Now the question is, what is an `AggregateException`? Do not worry! You will get familiar with it in the following section.

## Consider Using AggregateException

When you consider exception handling, we try to identify the potential error locations and place the try-catch (or try-catch-finally) block accordingly. There should be no confusion about this. However, in the case of task programming, exception handling is a challenging task. This is because, during the running state of an application, different tasks can throw different exceptions at various levels. You have also seen that if you are not careful enough, some of the exceptions can go unnoticed as well. And this is an obvious problem because any exception can cause a system to move into an inconsistent state. So, the question is, how can we handle exceptions in task programming?

There are indeed different approaches, but the first thing that you should note is that in the `System` namespace, there is a class called

`AggregateException` that inherits from the `Exception` class. Instead of using any other exception class, you should prefer this class. This class is specially designed to handle the exception handling scenarios when you deal with tasks. Microsoft states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library):

> *To propagate all the exceptions back to the calling thread, the Task infrastructure wraps them in an `AggregateException` instance. The `AggregateException` exception has an `InnerExceptions` property that can be enumerated to examine all the original exceptions that were thrown, and handle (or not handle) each one individually.*

This is why you will see that `AggregateException` is used heavily in the development code to consolidate multiple failures/errors in concurrent environments, particularly, when we use Task Parallel Library (TPL) and/or Parallel LINQ(PLINQ). This is why next time onward you will see me using `AggregateException` inside the `catch` block as well.

---

**Note** PLINQ is discussed in Chapter 5.

---

## Exception Management

How should we handle exceptions? Different programming model follows different strategies. For example, if you follow object-oriented programming (OOP), you'd like to use `try`, `catch`, and `finally` blocks. However, these are typically absent in functional programming (FP). In my other book *Introducing Functional Programming Using C#*, I had a detailed discussion on this topic. For now, we do not need to investigate those

details. Instead, let's simplify the overall strategies by putting them into the following categories:

- Handling possible exceptions in a single location
- Handling possible exceptions in multiple locations

## Handling Exceptions in a Single Location

In this section let's consider the first category, i.e., how to handle the possible exceptions in one place.

## Demonstration 4

Here is the complete program where I simulate the exceptional situations using some simple code. In this program, in addition to the `Product` class, you will see another class called `Database`. This class contains a method called `StoreData` that can throw an `InsufficientMemoryException` when a user tries to store more than 500 MB data. Here is the sample code for this:

```
class Database
{
    /// <summary>
    /// This method throws an exception when
    /// requested size is greater than 500MB
    /// </summary>
    /// <param name="sizeInMB"> The requested size for
    ///  allocation in MB</param>
    /// <returns> Confirming the storage</returns>
    /// <exception cref="InsufficientMemoryException"> The
    ///  exception is thrown when the <paramref
    /// name="sizeInMB" is greater than 500 </exception>
    public static string StoreData(int sizeInMB)
```

```
    {
        string allocation;
        if (sizeInMB > 500)
        {
             throw new InsufficientMemoryException($"Cannot
              store {sizeInMB} MB data.");
        }
        else
        {
            // Some code for allocation
            allocation = $"{sizeInMB} is allocated";
        }
        return allocation;
    }
}
```

Now I create two different tasks inside the main thread and simulate the code in a way that both tasks encounter exceptions. As discussed in the previous section, it will be sufficient for you to pass through the inner exceptions and display the error details. Go through the complete program now:

```
using static System.Console;
WriteLine("Exception handling demo.");

try
{
    var task1 = Task.Run(() => Product.CheckUser("abc"));
    var task2 = Task.Run(() => Database.StoreData(501));
    Task.WaitAll(task1, task2);
}
catch (AggregateException ae)
{
```

```
    foreach (Exception e in ae.InnerExceptions)
    {
        WriteLine($"Caught error: {e.Message}");
    }
}
// The Product class and the Database classes are not shown
// to avoid repetition.
```

## Output

Here is some sample output from this program:

```
Exception handling demo.
Caught error: Id: abc is invalid.
Caught error: Cannot store 501 MB data.
```

## Alternative Approaches

You have seen a simple approach to handling exceptions. Now let me show you two more approaches that can be used in a similar context.

### Alternative Approach 1

Notice the previous `catch` block. You can see that I used `ae.InnerExceptions` to display the errors in the output. Here is an alternative version where **I flatten the inner instances and then start traversing the exceptions** as follows:

```
catch (AggregateException ae)
{
    // Alternative approach-1
    var exceptions = ae.Flatten().InnerExceptions;
    foreach (Exception e in exceptions)
    {
```

```
        WriteLine($"Caught error: {e.Message}");
    }
}
```

## Alternative Approach 2

In the AggregateException class, you can see a method called Handle that has the following form:

```
public void Handle(Func<Exception, bool> predicate)
{
 // The method body is not shown
}
```

Using this method, you can invoke a handler on each exception contained in an AggregateException. For example, let's replace the foreach loop inside the catch block in Demonstration 4 with a new block of code as follows (I have shown all the different approaches that we have discussed so far using code comments for easy comparison):

```
catch (AggregateException ae)
{
    //// Initial approach
    // foreach (Exception e in ae.InnerExceptions)
    // {
    //     WriteLine($"Caught error: {e.Message}");
    // }

    //// Alternative approach-1
    // var exceptions = ae.Flatten().InnerExceptions;
    // foreach (Exception e in exceptions)
    // {
```

```
    //    WriteLine($"Caught error: {e.Message}");
    //}

    // Alternative approach-2
    ae.Handle(e =>
    {
        WriteLine($"Caught error: {e.Message}");
        return true;
    });
}
```

If you execute the program now, you will see the same output.

## Q&A Session

**Q2.2 I can see that you have thrown two different exceptions from two different tasks. If both tasks throw the same type of exception, how can I distinguish them?**

This is easy. You can associate the task IDs with the exception's `Source` property. For example, if you replace the following lines:

```
throw new UnauthorizedAccessException($"Id: {userId} is
 invalid.");
throw new InsufficientMemoryException($"Cannot store {sizeInMB}
MB data.");
```

with the following lines in the previous program:

```
throw new UnauthorizedAccessException($"Id: {userId} is
 invalid."){ Source = Task.CurrentId.ToString() };
throw new InsufficientMemoryException($"Cannot store {sizeInMB}
 MB data.") { Source = Task.CurrentId.ToString() };
```

and modify the `catch` block as follows:

```
catch (AggregateException ae)
{
    ae.Handle(e =>
    {
        WriteLine($"Caught error: {e.Message}
         Source: {e.Source}");
        return true;
    });
}
```

then you can get output like the following:

```
Exception handling demo.
Caught error: Id: abc is invalid. Source: 1
Caught error: Cannot store 501 MB data. Source: 2
```

Similarly, **if multiple tasks throw the same exception**, you can set the `Source` property to identify the source of the error.

## Demonstration 5

Here is a complete program to illustrate the idea:

```
WriteLine("Q&A on exception handling.");
try
{
    var task1 = Task.Run(
     () => throw new InsufficientMemoryException($"Cannot
           store 500 MB data."){ Source = "task1" });
    var task2 = Task.Run(
     () => throw new InsufficientMemoryException($"Cannot
           store 500 MB data."){ Source = "task2" });
```

```
    Task.WaitAll(task1, task2);
}
catch (AggregateException ae)
{
    ae.Handle(e =>
    {
        WriteLine($"Caught error: {e.Message} [From
         {e.Source}]");
        return true;
    });
}
```

## Output

Here is the sample output:

```
Q&A on exception handling.
Caught error: Cannot store 500 MB data. [From task1]
Caught error: Cannot store 500 MB data. [From task2]
```

## Handling Exceptions in Multiple Locations

I assume that you have gotten the idea of how to handle multiple exceptions. This is OK and probably the most common approach. Next, I will show you a mechanism where you can handle one part of the aggregate exception in one place and the remaining part in another place. More specifically, you propagate this remaining part of the exception up to the hierarchy and handle it there. There is a reason for this: I want you to show the effectiveness of the `Handle` method.

First, see the following code. This code fragment indicates that you catch the probable set of exceptions but handle only one of them: `InsufficientMemoryException`.

```
// Some code before
catch (AggregateException ae)
{
 ae.Handle(
   e =>
   {
     if (e is InsufficientMemoryException)
     {
       WriteLine($"Caught error: {e.Message}");
       return true;
     }
     else
     {
       return false;
     }
   });
}
```

By returning `true`, you indicate that this particular exception is handled. More specifically, this code fragment says that you like to handle only the `InsufficientMemoryException` but no other exceptions in this location.

## Demonstration 6

For example, in the upcoming program, the main thread calls the `InvokeTasks` method that, in turn, creates and runs three new tasks as follows:

```
var task1 = Task.Run(() => Product.CheckUser("abc"));
var task2 = Task.Run(() => Database.StoreData(501));
var task3 = Task.Run( () => throw new DllNotFoundException
  ("the dll is missing!"));
```

The first two tasks are already shown in the previous demonstration. You know that these tasks will raise exceptions. *The third task, named task3, is added for the sake of discussion so that you do not assume that you need to handle an equal number of tasks in each location.*

You'll see that I catch all the possible sets of exceptions inside `InvokeTasks` but handle only one of them: `InsufficientMemoryException`. As a result, the remaining exceptions will be passed up to the calling hierarchy. So, I have handled them inside the `Main` method aka main thread. Let's go through the complete program now.

---

**Note** I remind you that I have heavily used top-level statements and enabled implicit `using` statements for the C# projects in this book. These features came in C# 9.0 and C# 10.0, respectively. This is why you do not see a method named `Main` in this program.

---

```
using static System.Console;
WriteLine("Exception handling demo.");
try
{
    InvokeTasks();
}
catch (AggregateException ae)
{
    ae.Handle(e =>
    {
        WriteLine($"Caught error inside Main(): {e.Message}");
        return true;
    });
}
```

```
static void InvokeTasks()
{
    try
    {
        var task1 = Task.Run(() => Product.CheckUser("abc"));
        var task2 = Task.Run(() => Database.StoreData(501));
        var task3 = Task.Run(() => throw new
           DllNotFoundException("the dll is missing!"));
        Task.WaitAll(task1, task2, task3);
    }
    catch (AggregateException ae)
    {
        // Handling only InsufficientMemoryException, others
        // will be propagated up to the hierarchy
        ae.Handle(
           e =>
           {
               if (e is InsufficientMemoryException)
               {
                   WriteLine($"Caught error inside
                    InvokeTasks(): {e.Message}");
                   return true;
               }
               else
               {
                   return false;
               }
           });
    }
}
```

```
// There was no change inside the "Product" class and the
// "Database" class. So, they are not shown again to avoid
// repetitions.
```

---

**Note** You can download the project `Demo6_HandlingExceptionsSeparately` from the Apress website to see and execute the complete program.

---

## Output

Here is some sample output from this program:

```
Exception handling demo.
Caught error inside InvokeTasks(): Cannot store 501 MB data.
Caught error inside Main(): Id: abc is invalid.
Caught error inside Main(): the dll is missing!
```

## Final Suggestion

I hope you have gotten an idea of how to handle exceptions in a parallel execution environment. I'd like to share one final thought with you: if you do not handle exceptions within tasks, you should try to handle them as close as possible to those places where you wait for the task completion and/or retrieve the result of the task invocation.

# Understanding Cancellations

If you want to open a fixed deposit (FD) in India, banks give you two options: callable FD and non-callable FD. Usually, the interest rates for non-callable FDs are slightly higher than their counterpart. This is because

you are not allowed to cancel the FD and withdraw the amount before the specified date. But you know that in a crisis, a customer may need to cancel the FD. So, if the customer does not have sufficient backup, he may opt for a callable FD. Similar situations are observed in programming as well. You may need to work with several tasks, and while those tasks are running, you may need to cancel one or more of them due to various reasons. Some of the examples include terminating a long-running process or releasing a critical resource. This is why you may like to work with cancellable tasks. Let's see how to cancel a task.

## Different Ways of Cancellation

To support possible cancellations, this time onward you'll see me using cancellation tokens. To generate such a token, you need the instances of the `CancellationTokenSource` and `CancellationToken`. You may see the following lines of code in this context:

```
CancellationTokenSource tokenSource = new();
CancellationToken token = tokenSource.Token;
```

Obviously, using the `var` keyword, you can see a slight variation as follows:

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
```

Next, you pass this token to the intended task. Earlier, you saw (in Figure 1-1 in Chapter 1) that the `Task` constructor has several overloaded versions. Some of them accept a `CancellationToken` instance as a method parameter. Here is an example:

```
public Task(Action action, CancellationToken cancellationToken);
```

The StartNew method of the TaskFactory class has the overloaded versions as well, and some of them also accept a CancellationToken instance as a method parameter. Here is an example:

```
public Task StartNew(Action action, CancellationToken
 cancellationToken);
```

Similarly, the Run method of the Task class has similar overloads.

These constructs give you a clue on how to pass a CancellationToken instance to a task. Shortly, you'll see me using the following version:

```
public static Task Run(Action action, CancellationToken
 cancellationToken)
```

In the upcoming demonstration, I created a task that can keep printing the numbers from 0 to 99. Nowadays computer processors are very fast. So, this task can finish its execution very fast. To prevent this, I impose a short sleep after it prints a number. I also opt for the support for cancellation while it executes. So, I create a CancellationToken instance, called token, and pass it as follows:

```
var printTask=Task.Run
    (
      () =>
      {
          // A loop that runs 100 times
          for (int i = 0; i < 100 ; i++)
          {
              // Some other code is skipped here
              WriteLine($"{i}");
              // Imposing the sleep to make some delay
              Thread.Sleep(500);
          }
      }, token
    );
```

However, you must remember the following guidelines from Microsoft (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-cancel-a-task-and-its-children):

> *The calling thread does not forcibly end the task; it only signals that cancellation is requested. If the task is already running, it is up to the user delegate to notice the request and respond appropriately.*

---

**Note** This prior message indicates that it is possible that by the time a calling thread raises a cancellation request, the running task finishes its execution. So, if you want to cancel a running task, you should raise the cancellation request as soon as possible.

---

Let me show you some cancellation techniques.

## Initial Approach

In the first approach, you can use an `if` condition to evaluate whether the cancellation request is raised. If so, you can do something before you cancel the task. For example, you can introduce a message saying that this task is going to cancel itself (and clean up the resources, if required), and then you can put a `break` or `return` statement. Probably, most of us are aware of this kind of "soft exit" mechanism. Let me show you an example.

## Demonstration 7

The following demonstration shows you a sample where I added a new code block in the previous code segment and made it bold for your reference:

```
using static System.Console;

WriteLine("Simple cancellation demonstration.");
```

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

var printTask = Task.Run
 (
  () =>
  {
      // A loop that runs 100 times
      for (int i = 0; i < 100; i++)
      {
          if (token.IsCancellationRequested)
          {
              WriteLine("Cancelling the print activity.");
              // Do some cleanups, if required
              return;
          }
          WriteLine($"{i}");
          // Imposing the sleep to make some delay
          Thread.Sleep(500);
      }
  }, token
);

WriteLine("Enter c to cancel the task.");
char ch = ReadKey().KeyChar;
if (ch.Equals('c'))
{
    WriteLine("\nTask cancellation requested.");
    tokenSource.Cancel();
}
// Wait till the task finishes the execution
while (!printTask.IsCompleted) { }
```

```
WriteLine($"The final status of printTask is: {printTask.Status}");
WriteLine("End of the main thread.");
```

**POINT TO NOTE**

This program uses the following line:

```
while (!someTask.IsCompleted) { }
```

Microsoft suggests you avoid this kind of polling in the production code as it is very inefficient (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library). So, a better solution can be made with `printTask.Wait();`. But if I use the `Wait()` method, I need to guard the possible exceptions using `try-catch` blocks. To avoid this, I have used this line of code in this program and the next few demonstrations because I wanted to focus purely on cancellations, instead of mixing it up with exceptions.

## Output

Here is some sample output:

```
Simple cancellation demonstration.
Enter c to cancel the task.
0
1
2
c
Task cancellation requested.
Cancelling the print activity.
The final status of printTask is: RanToCompletion
End of the main thread.
```

## Q&A Session

**Q2.3 In the previous demonstration, you canceled the task, but in the output, the final status of the task was displayed as `RanToCompletion`, instead of `Canceled`. Is this a bug?**

No. Let's see what Microsoft says about it. The online documentation states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation):

> *In the Task classes, cancellation involves cooperation between the user delegate, which represents a cancelable operation, and the code that requested the cancellation. A successful cancellation involves the requesting code calling the `CancellationTokenSource.Cancel` method and the user delegate terminating the operation in a timely manner. You can terminate the operation by using one of these options:*
>
> - By returning from the delegate. In many scenarios, this option is sufficient. However, a task instance that's canceled in this way transitions to the `TaskStatus.RanToCompletion` state, not to the `TaskStatus.Canceled` state.
> - By throwing an `OperationCanceledException` and passing it the token on which cancellation was requested. The preferred way to perform this is to use the `ThrowIfCancellationRequested` method. A task that's canceled in this way transitions to the `Canceled` state, which the calling code can use to verify that the task responded to its cancellation request.

The first bullet point is easy to understand and justifies the final status of `printTask` in Demonstration 7. In the next section, I'll show you the other approach where you will notice that the final status is `Canceled`.

## Alternative Approach

Let's see the alternative ways of cancellations as well. In a similar context, developers often like to throw an `OperationCanceledException` exception as follows:

```
if (token.IsCancellationRequested)
 {
     WriteLine("Cancelling the print activity.");
     throw new OperationCanceledException(token);
 }
```

## Demonstration 8

It is time for another demonstration. You can update the task definition in Demonstration 7 as follows:

```
var printTask = Task.Run
 (
  () =>
  {
      // A loop that runs 100 times
      for (int i = 0; i < 100; i++)
      {
          // Approach-1
          // if (token.IsCancellationRequested)
          // {
          //     WriteLine("Cancelling the print activity.");
          //     // Do some cleanups, if required
          //     return;
          // }
```

```
            // Approach-2
            if (token.IsCancellationRequested)
            {
                WriteLine("Cancelling the print activity.");
                // Do some cleanups, if required
                throw new OperationCanceledException(token);
            }

            WriteLine($"{i}");
            // Imposing the sleep to make some delay
            Thread.Sleep(500);
        }
    }, token
);
```

Now execute the program again.

## Output

Here is a sample output. Notice that in Demonstration 7, the final status of the task was `RanToCompletion`, but in this demonstration, it appears as `Canceled`.

```
Simple cancellation demonstration.
Enter c to cancel the task.
0
1
2
c
Task cancellation requested.
Cancelling the print activity.
The final status of printTask is: Canceled
End of the main thread.
```

## Shortening the Code

Microsoft says that the ThrowIfCancellationRequested method is the functional equivalent of the following lines (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken.throwifcancellationrequested?view=net-8.0):

```
if (token.IsCancellationRequested)
    throw new OperationCanceledException(token);
```

This implies that you can use the ThrowIfCancellationRequested method for the following two things:

- You can check whether a cancellation request is raised.
- You can throw the OperationCanceledException exception when such a request is raised.

This is why you can shorten the code as follows:

```
var printTask = Task.Run
 (
  () =>
  {
      // A loop that runs 100 times
      for (int i = 0; i < 100; i++)
      {
          // Approach-3
          token.ThrowIfCancellationRequested();
          WriteLine($"{i}");
          // Imposing the sleep to make some delay
          Thread.Sleep(500);
      }
  }, token
);
```

This is a common and widely used approach. Normally, I prefer to use this approach. If you like to do some cleanups before this call, you can write something like the following as well:

```
if (token.IsCancellationRequested)
{
    // Do some cleanups, if required
   token.ThrowIfCancellationRequested();
}
```

## Q&A Session

**Q2.4 In Demonstration 7, you simply did a soft exit and got the final task status as `RanToCompletion`, whereas in Demonstration 8, the final task status was canceled. I understand that this is a design decision, but I'd like to know your thoughts on them.**

Normally, I'd like to follow the approach that is shown in Demonstration 8. This is because in an enterprise application, we normally deal with several tasks, and we often need to understand the log/output. In those cases, I can go through the log to understand which task was canceled. But if you simply exit from the method without doing anything, there is no such record left for you.

## One Final Note

At the time of this writing, the `OperationCanceledException` class has 7 overloaded constructors that may take 0, 1 or 2 parameters. You can use them to initialize a new instance of `OperationCanceledException` as per your needs. Here I include some of them for your reference:

- `OperationCanceledException(CancellationToken)`: Initializes a new instance with a cancellation token. (You have already seen this in Demonstration 8).

- `OperationCanceledException(String)`: Initializes a new instance with a specified error message.
- `OperationCanceledException()`: Initializes a new instance with the system-supplied error message.

See that you can avoid passing a `CancellationToken` instance to initialize a new instance of `OperationCanceledException` Or, you can instantiate it with a different token. In such cases, you get the final status as `Faulted` (instead of `Canceled`).

To experiment with this, let's create a new token, say `token2`, and update the `if` block in Demonstration 8 as follows:

```
// Approach-2
if (token.IsCancellationRequested)
{
  WriteLine("Cancelling the print activity.");
  // Do some cleanups, if required
  throw new OperationCanceledException(token2);
}
```

Execute the program again. Notice that the final status appears as `Faulted`, but not `Canceled`. Here is a sample for you:

```
Simple cancellation demonstration.
Enter c to cancel the task.
0
1
2
3
c
Task cancellation requested.
Cancelling the print activity.
The final status of printTask is: Faulted
End of the main thread.
```

## Q&A Session

**Q2.5 Why does the output show the final status `Faulted` instead of `Canceled`?**

This is a design decision. The documentation states the following (https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation):

> *If the token's `IsCancellationRequested` property returns false or if the exception's token doesn't match the Task's token, the `OperationCanceledException` is treated like a normal exception, causing the Task to transition to the `Faulted` state. The presence of other exceptions will also cause the Task to transition to the `Faulted` state.*

# Monitoring Task Cancellation

Have you noticed the output of Demonstration 7 and Demonstration 8? In both cases, you saw the following line: `Cancelling the print activity.`. I used this line to monitor the canceled task before the cancellation operation. Interestingly, there are alternatives. Let's see some of them.

## Using Register

You can subscribe to an event notification. For example, in the following code, I register a delegate that will be called when the token is canceled:

```
token.Register(
    () =>
    {
        WriteLine("Cancelling the print activity.
        [Using event subscription]");
        // Do something else, if you want
    }
  );
```

## Using WaitHandle.WaitOne

Let me show you one more approach that is relatively complicated compared to the previous one. However, this will also give you an idea about how to monitor task cancellation. The documentation describes `WaitHandle`'s `WaitOne` method as follows (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.waithandle.waitone?view=net-8.0):

> *Blocks the current thread until the current WaitHandle receives a signal.*

The `WaitOne` method has many overloads. In the upcoming demonstration, I'll show you the simplest form that does not require you to pass any argument. The basic idea is that the current thread will consider a token and wait until someone cancels it. As soon as someone invokes the cancellation, the blocking function call will be released. This is why I can launch another task from the calling thread as follows:

```
Task.Run(
    () =>
    {
        token.WaitHandle.WaitOne();
        WriteLine("Cancelling the print activity.
         [Using WaitHandle]");
        // Do something else, if you want
    }
);
```

Notice that it is similar to subscribing to an event notification, because here also you wait for the cancellation to occur. This is why I have written a similar statement in this code block.

## Demonstration 9

It is time for another demonstration where I show you two approaches to monitor the cancellation operation. Notice the key changes in bold:

```
using static System.Console;

WriteLine("Monitoring the cancellation operation.");

var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

token.Register(
    () =>
    {
        WriteLine("Cancelling the print activity.[Using event
         subscription]");
        // Do something else, if you want
    }
  );

var printTask = Task.Run
 (
  () =>
  {
      // A loop that runs 100 times
      for (int i = 0; i < 100; i++)
      {
          // Approach-3
          token.ThrowIfCancellationRequested();
          WriteLine($"{i}");
          // Imposing the sleep to make some delay
          Thread.Sleep(500);
      }
```

```
  }, token
);

Task.Run(
    () =>
    {
        token.WaitHandle.WaitOne();
        WriteLine("Cancelling the print activity.[Using
         WaitHandle]");
        // Do something else, if you want
    }
);

WriteLine("Enter c to cancel the task.");
char ch = ReadKey().KeyChar;
if (ch.Equals('c'))
{
    WriteLine("\nTask cancellation requested.");
    tokenSource.Cancel();
}
// Wait till the task finishes the execution
while (!printTask.IsCompleted) { }
WriteLine($"The final status of printTask is: {printTask.Status}");
WriteLine("End of the main thread.");
```

## Output

Here is some sample output. Notice the changes in bold.

```
Monitoring the cancellation operation.
Enter c to cancel the task.
0
```

```
1
2
c
Task cancellation requested.
Cancelling the print activity.[Using WaitHandle]
Cancelling the print activity.[Using event subscription]
The final status of printTask is: Canceled
End of the main thread.
```

## Cancelling Child Tasks

I assume you understand task cancellations now. I'd like to remind you that Demonstration 8 in Chapter 1 showed you how to cancel a child task to answer Q1.8. That is why I won't repeat the discussion here.

## Q&A Session

**Q2.6 I understand that to provide support for cancellation, I need to create a `CancellationTokenSource` instance and get a token from it. However, I may need to support several cancellation strategies in a program. How can I implement the concept?**

Good question. You may need to work with several tokens. The upcoming demonstration will give you the idea.

## Managing Multiple Cancellation Tokens

An application can indeed be canceled due to various reasons. In such a case, you can use multiple tokens and provide the necessary logic. In this context, you can use the `CreateLinkedTokenSource` method. Let's see an example.

In the following demonstration, you'll see two different tokens as follows:

```
var normalCancellation = new CancellationTokenSource();
var tokenNormal = normalCancellation.Token;

var unexpectedCancellation = new CancellationTokenSource();
var tokenUnexpected = unexpectedCancellation.Token;
```

Once the tokens are created, I pass them to the `CreateLinkedTokenSource` method as follows:

```
var compositeToken = CancellationTokenSource.
 CreateLinkedTokenSource(tokenNormal,tokenUnexpected);
```

The idea is that you can cause a cancellation using either `normalCancellation` or `unexpectedCancellation`.

You may note that the `CreateLinkedTokenSource` method has different overloads, and you can pass more tokens if required. Remember that the core idea is the same: you can cancel any of these tokens to make the final task status `Canceled`.

## Demonstration 10

In the following program, a user can trigger a normal cancellation. But you can also observe an unexpected/emergency cancellation as well. To mimic an emergency cancellation, I rely on a generated random number. If the number is 5, the unexpected cancellation will be triggered. Here is the complete program to demonstrate the idea:

```
using static System.Console;

WriteLine("Monitoring the cancellation operation.");

var normalCancellation = new CancellationTokenSource();
var tokenNormal = normalCancellation.Token;

var unexpectedCancellation = new CancellationTokenSource();
```

```
var tokenUnexpected = unexpectedCancellation.Token;

tokenNormal.Register(
    () =>
    {
        WriteLine(" Processing a normal cancellation.");
        // Do something else, if you want
    }
  );

tokenUnexpected.Register(
    () =>
    {
        WriteLine(" Processing an unexpected cancellation.");
        // Do something else, if you want
    }
  );

var compositeToken = CancellationTokenSource.CreateLinkedTokenSource(
 tokenNormal,
 tokenUnexpected
);

var printTask = Task.Run
 (
  () =>
  {
      // A loop that runs 100 times
      for (int i = 0; i < 100; i++)
      {
          compositeToken.Token.ThrowIfCancellationRequested();
          WriteLine($"{i}");
```

```
            // Imposing sleep to make some delay
            Thread.Sleep(500);
        }
    }, compositeToken.Token
);

int random = new Random().Next(1, 6);
// A dummy logic to mimic an emergency cancellation
if (random == 5)
    unexpectedCancellation.Cancel();
else
{
    WriteLine("Enter 'c' for a normal cancellation ");
    char ch = ReadKey().KeyChar;
    if (ch.Equals('c'))
    {
        WriteLine("\nTask cancellation requested.");
        normalCancellation.Cancel();
    }
}

// Wait till the task finishes the execution
while (!printTask.IsCompleted) { }
WriteLine($"The final status of printTask is: {printTask.Status}");
WriteLine("End of the main thread.");
```

## Output

Here is some sample output after a user presses C to mimic a normal cancellation:

```
Monitoring the cancellation operation.
Enter 'c' for a normal cancellation
```

```
0
1
2
c
Task cancellation requested.
 Processing a normal cancellation.
The final status of printTask is: Canceled
End of the main thread.
```

Here is some more sample output when an emergency cancellation was triggered:

```
Monitoring the cancellation operation.
0
 Processing an unexpected cancellation.
The final status of printTask is: Canceled
End of the main thread.
```

## Microsoft's Note for Visual Studio Users

If you are a Visual Studio user and write programs that deal with multiple cancellation requests, I want you to remember the following note from Microsoft (see https://learn.microsoft.com/en-us/dotnet/standard/threading/how-to-listen-for-multiple-cancellation-requests):

> *When "Just My Code" is enabled, Visual Studio in some cases will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue from it.*

It keeps saying the following:

> *To prevent Visual Studio from breaking on the first error, just uncheck the "Just My Code" checkbox under Tools, Options, Debugging, General.*

# Organizing Exceptions and Cancellations

In Chapter 1, you learned that there are three possible final states of a task: `RanToCompletion`, `Canceled`, and `Faulted`. In this chapter, you learned about cancellations and exceptions (particularly, we talked about `AggregateException` and `OperationCanceledException`). However, up until now, I discussed exceptions and cancellations in separate sections. This is why typical `try-catch` blocks were absent when I discussed cancellations. But typical software must be prepared for exceptions and cancellations. So, it is time to consolidate all this knowledge in a single place. Let's analyze some case studies.

## Case Study 1: Using Wait()

The following program simulates a money transfer process. It also shows the progress status in the output. To simplify everything, let's assume that a money transfer takes five seconds. So, after every second, I increase the progress status by 20%.

In the upcoming program, there is a `try` block followed by two `catch` blocks. Why? Instead of catching all exceptions in a single `catch` block, I'd like to use separate `catch` blocks to handle the possible scenarios of exceptions and cancellations. One of these `catch` blocks handles `AggregateException`, and another `catch` block handles `OperationCanceledException`.

### Demonstration 11

Here is the complete program:

```
using static System.Console;
WriteLine("Handling cancellations and exceptions.");
CancellationTokenSource cts = new();
CancellationToken token = cts.Token;
```

```
var transferMoney = Task.Run(
 () =>
 {
     WriteLine($"Initiating the money transfer.");
     int progressBar = 0;
     WriteLine("Press c to cancel within 5 sec.");
     // Assuming the task will take 5 seconds.
     // So, after every second, we'll increase
     // the progress by 20%
     for (int i = 0; i < 5; i++)
     {
         token.ThrowIfCancellationRequested();
         Thread.Sleep(1000);
         progressBar += 20;
         WriteLine($"Progress:{progressBar}%");
     }
     return "The money transfer is completed.";
 }, token);

var input = ReadKey().KeyChar;
if (input.Equals('c'))
{
    WriteLine("\nCancellation is requested.");
    cts.Cancel();
}
try
{
    transferMoney.Wait();
}
catch (AggregateException ae)
{
    ae.Handle(e =>
    {
```

```
        WriteLine($"Caught error:  {e.Message}");
        return true;
    });
}

catch (OperationCanceledException oce)
{
    WriteLine($"Caught error due to cancellation: {oce.Message}");
}

if (transferMoney.Status == TaskStatus.RanToCompletion)
{
    WriteLine(transferMoney.Result);
}

WriteLine($"Current status: {transferMoney.Status}");
WriteLine("Thank you, visit again!");
```

## Output

Let me show you some possible outputs. Notice the important lines in bold in these outputs.

**Case 1:** In this case, you do not cancel the task. Here is some sample output:

```
Handling cancellations and exceptions.
Initiating the money transfer.
Press c to cancel within 5 sec.
Progress:20%
Progress:40%
Progress:60%
Progress:80%
Progress:100%
The money transfer is completed.
Current status: RanToCompletion
Thank you, visit again!
```

**Case 2:** Here is some sample output from the program when you try to cancel the task:

```
Handling cancellations and exceptions.
Initiating the money transfer.
Press c to cancel within 5 sec.
Progress:20%
Progress:40%
Progress:60%
c
Cancellation is requested.
Progress:80%
Caught error:  A task was canceled.
Current status: Canceled
Thank you, visit again!
```

## Case Study 2: Using Wait(token)

Let's replace the `try` block in the previous program with the following one:

```
// There is no change in the previous code
try
{
    //transferMoney.Wait();
    transferMoney.Wait(token);
// There is no change in the remaining code
```

Let's execute this modified program and analyze the previous cases again. *First, if you do not cancel the task, you will not observe any change in the output. However, if you initiate a cancellation, there are some differences.* Let me show you some sample output:

```
Handling cancellations and exceptions.
Initiating the money transfer.
```

```
Press c to cancel within 5 sec.
Progress:20%
Progress:40%
Progress:60%
c
Cancellation is requested.
Caught error due to cancellation: The operation was canceled.
Current status: Running
Thank you, visit again!
```

You can see that this time the `catch` block for `AggregateException` could not catch `OperationCanceledException`; this is why I used two different `catch` blocks in this program.

## Q&A Session

**Q2.7 Why does the previous output show the final status `Running` instead of `Canceled`?**

The `Wait(token)` differs from `Wait()`. In the case of `Wait(token)`, the wait terminates if a cancellation token is canceled before the task is completed. In this case, the main thread exited early. So, to see the final status of the task as `Canceled`, you can introduce the following code (shown in bold) in the following location:

```
// There is no change in the previous code.

// Wait till the task finishes the execution
while (!transferMoney.IsCompleted) { }
WriteLine($"Current status:{transferMoney.Status}");

// There is no change in the remaining code.
```

Here is some sample output after this change:

```
Handling cancellations and exceptions.
Initiating the money transfer.
Press c to cancel within 5 sec.
Progress:20%
Progress:40%
Progress:60%
c
Cancellation is requested.
Caught error due to cancellation: The operation was canceled.
Progress:80%
Payment processing status: Canceled
Thank you, visit again!
```

**POINT TO NOTE**

When you examine this in detail, you'll see that `Wait()` can throw only `AggregateException`, whereas `Wait(CancellationToken cancellationToken)` is cancellable and can raise `OperationCanceledException`. If interested, you can see a discussion of this topic at https://stackoverflow.com/questions/77833724/why-the-catch-block-of-aggregateexception-was-not-sufficient-to-handle-cancellat/77833858#77833858.

# Handling I/O-Bound Tasks

In addition to running the essential tasks, sometimes you'll want to run an additional task to monitor the overall activity. At a later stage, you may provide a choice: whether the user likes to retrieve the details of the activity. In a traditional scenario, a user supplies some specific input(s) to

know the details. You may think that this is a common activity. Wait! There is something more to add: since it purely depends on the user's choice, you would like to run this additional task without blocking the current thread. Can you design such an application? The following section will show you an implementation strategy.

## Using TaskCompletionSource

You can consider using the `TaskCompletionSource<TResult>` class to implement the idea. Using it you can a create task out of any operation that will be completed in the future. Microsoft describes this class as follows (see `https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1?view=net-8.0`):

> *Represents the producer side of a Task<TResult> unbound to a delegate, providing access to the consumer side through the Task property.*

More specifically, behind the scenes, using this class, you get a "slave" task that you can manually drive when the operation finishes (or faults). It can be ideal for such a type of I/O-bound work where you reap all the benefits of using a task without blocking the calling thread.

Now the question is: how do you use this class? First, you need to instantiate it. Once you instantiate this class, you will get some built-in methods that can serve your purpose. Figure 2-4 shows the details of this class in Visual Studio.

```
using System.Collections.Generic;

namespace System.Threading.Tasks
{
    public class TaskCompletionSource<TResult>
    {
        public TaskCompletionSource();
        public TaskCompletionSource(object? state);
        public TaskCompletionSource(TaskCreationOptions creationOptions);
        public TaskCompletionSource(object? state, TaskCreationOptions creationOptions);

        public Task<TResult> Task { get; }

        public void SetCanceled();
        public void SetCanceled(CancellationToken cancellationToken);
        public void SetException(IEnumerable<Exception> exceptions);
        public void SetException(Exception exception);
        public void SetResult(TResult result);
        public bool TrySetCanceled();
        public bool TrySetCanceled(CancellationToken cancellationToken);
        public bool TrySetException(IEnumerable<Exception> exceptions);
        public bool TrySetException(Exception exception);
        public bool TrySetResult(TResult result);
    }
}
```

***Figure 2-4.*** *The TaskCompletionSource class details*

Before you see the code examples, I suggest that you note the following points:

- Figure 2-4 confirms that the method names start either with `Set` or with `TrySet`. The first category returns `void`, and the second category returns `bool`.
- When you call any of these methods, the task moves into any of the final states: `RanToCompletion`, `Faulted`, or `Canceled`.
- You should call the first category (i.e., where the method names start with the word `Set`) exactly once; otherwise, you'll see exceptions.

To illustrate the previous bullet point, let me give you an example: in the upcoming program, there is a custom class called `Job`. I created an instance of it, called `job`, and used it in the task definition as follows (notice that the `SetResult` method is called exactly once):

```
var backgroundTask = Task.Run(
    () =>
    {
        // Some other codes before this line are not shown
        // to focus on the main point of discussion
        // taskCompletionSource.SetResult(job);
        // The following line will cause an exception now
        // taskCompletionSource.SetResult(job);

    });
```

Now if you uncomment the commented-out line in the previous code and wait for this task to finish (for example, using the line `backgroundTask.Wait();`), you will see the following exception in the final output:

```
Unhandled exception. System.AggregateException: One or more
errors occurred. (An attempt was made to transition a task to a
final state when it had already completed.)
// The remaining details are not shown
```

To avoid this kind of error, I prefer to use the `TrySetResult` method instead of the `SetResult` method. The reason is obvious: it returns a Boolean, i.e., either `true` or `false`.

---

**Note** In the `Demo12_UsingTaskCompletionSource` project, I kept the erroneous code block as commented code for your experimentation/reference. You can download it from the Apress website.

---

## Demonstration 12

To understand the following demonstration, go through the following points:

- There is a `Job` class in this program. To create an instance of it, I pass a job identification number and the name of the executor. Passing the executor's name was optional for me.
- At the beginning of the program, I created an instance of the `TaskCompletionSource<Job>` class, called `taskCompletionSource`. As a result, I can use its `Task` property in a later stage.
- The user of this application can get the details of the created `Job` instance by entering the character `y` to get the details. In such a case, the result will be collected through a task called `backgroundTask` that completes `taskCompletionSource.Task`.
- If the user enters any other character, the program completes execution without showing the job details.

Let's see the complete program now:

```
using static System.Console;

TaskCompletionSource<Job> taskCompletionSource = new();
Task<Job> collectInfoTask = taskCompletionSource.Task;

WriteLine($"Starts processing a job.");
Job job = new(1)
{
    Executor = "Roy"
};
```

```
// Do something else if required

// Starting a background task that will complete
// taskCompletionSource.Task
var backgroundTask = Task.Run(
() =>
{
 WriteLine(" Monitoring the activity before setting the result.");
 // Imposing some forced delay before setting the
 // result to mimic real-world
 Thread.Sleep(3000);
 bool setResultStatus= taskcompletionSource.TrySetResult(job);
});

// Imposing a forced delay so that the background task can
// start running before executing the rest of the code
Thread.Sleep(1000);

WriteLine("Press 'y' to get the details.");
var input = ReadKey();
if (input.KeyChar == 'y')
{
  WriteLine(collectInfoTask.Result);
  // Same as:
  // WriteLine(taskCompletionSource.Task.Result);
}
WriteLine("\nThank you!");

internal class Job
{
    public int JobNumber { get; init; }
    public string Executor { get; set; }
```

```
    public Job(int jobNumber, string executor = "Anonymous")
    {
        JobNumber = jobNumber;
        Executor = executor;
    }
    public override string ToString() =>
     $"\n{Executor} executed the job number {JobNumber}";
}
```

## Output

Here is some sample output:

**Case-1( The user opt for the detail):**

```
Starts processing a job.
 Monitoring the activity before setting the result.
Press 'y' to get the details.
y
Roy executed the job number 1

Thank you!
```

**Case-2( The user does not opt for the detail):**

```
Starts processing a job.
 Monitoring the activity before setting the result.
Press 'y' to get the details.
n
Thank you!
```

## Q&A Session

**Q2.8 I can see that you have used a property and a constructor to instantiate a `Job` instance. Is there any specific reason behind this design?**

Yes. You can indeed opt for any of them. But notice that I have used the `init` accessor in this example. About this accessor, Microsoft says the following (see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/init):

> *In C# 9 and later, the init keyword defines an accessor method in a property or indexer. An init-only setter assigns a value to the property or the indexer element only during object construction. This enforces immutability, so that once the object is initialized, it can't be changed again.*

This is why the job number cannot be changed in this program, but its executor can be reassigned. Fixing the job number for a particular job, I promote immutability. In a multithread environment, promoting immutability is a good idea.

**Q2.9 How does immutability fit into a multithreaded programming environment?**

Immutable types are thread-safe. They prevent nasty bugs from implicit dependencies, destructive updates, and state mutations. Therefore, in a multithreaded environment, they make your programming life easy.

In addition, immutable types can help you avoid side effects and temporal coupling as well. Indeed, I am not focusing on functional programming in this book. So, for the simple demonstrations, I do not care about them. I simply wanted you to keep these notes in your mind using this program.

# Exercises

Check your understanding by attempting the following exercises:

**REMINDER**

As said before for Chapter 1's exercises, for all code examples, the Implicit Global Usings setting was enabled in Visual Studio. This is why you will not see me mentioning some other namespaces that were available by default. You can safely assume that all other necessary namespaces are available for these code segments. The same comment applies to all exercises in this book as well.

**E2.1** If you execute the following code, can you predict the output?

```
using static System.Console;
WriteLine("Exercise 2.1");

try
{
    int b = 0;
    Task<int> value = Task.Run(() => 10 /b);
}
catch (Exception e)
{
    WriteLine($"Caught error: {e.Message}");
}
WriteLine("End");
```

**E2.2** If you execute the following code, can you predict the output?

```
using static System.Console;
WriteLine("Exercise 2.2");
try
```

```
{
    int b = 0;
    Task<int> value = Task.Run(() => 10 / b);
    WriteLine(value.Result);
}
catch (Exception e)
{
    WriteLine($"Encountered with {e.GetType()}");
}
WriteLine("End");
```

**E2.3** If you execute the following code, can you predict the output?

```
using static System.Console;
WriteLine("Exercise 2.3");

try
{
    var task1 = Task.Run(
      () => throw new InvalidDataException("invalid data"));
    var task2 = Task.Factory.StartNew(
      () => throw new OutOfMemoryException("insufficient
            memory"));
    Task.WaitAll(task1, task2);
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        WriteLine($"Caught error: {e.Message}");
    }
}
```

**E2.4** If you execute the following code, can you predict the output?

```
using static System.Console;
WriteLine("Exercise 2.4");

try
{
    var task1 = Task.Run( () => throw new
     InvalidDataException("invalid data"));
    var task2 = Task.Run(() => throw new
     OutOfMemoryException("insufficient  memory"));
    task1.Wait();
    task2.Wait();
    WriteLine("End");
}
catch (AggregateException ae)
{
    ae.Handle(e =>
        {
            if (e is InvalidDataException |
                e is OutOfMemoryException )
            {
                WriteLine($"Caught error: {e.Message}");
                return true;
            }
            return false;
        }
    );
}
```

**E2.5** If you execute the following code, can you predict the output?

```
using static System.Console;
WriteLine("Exercise 2.5 and Exercise 2.6");
```

```
try
{
    DoSomething();
}
catch (AggregateException ae)
{
    ae.Handle(
        e =>
        {
            WriteLine($"Caught inside main: {e.Message}");
            return true;
        }
    );
}
static void DoSomething()
{
    try
    {
      var task1 = Task.Run(() => throw new
       InvalidDataException("invalid data"));
      var task2 = Task.Run(() => throw new
       OutOfMemoryException("insufficient memory"));
        // For Exercise 2.5
        Task.WaitAll(task1, task2);
        // For Exercise 2.6
        // task1.Wait();
        // task2.Wait();
    }
    catch (AggregateException ae)
    {
        ae.Handle(
            e =>
```

```
            {
                if (e is InvalidDataException)
                {
                    WriteLine($"Caught inside DoSomething:
                     {e.Message}");
                    return true;
                }
                else
                {
                    return false;
                }
            }
        );
    }
}
```

**E2.6** Predict the output when you replace the following line:

```
Task.WaitAll(task1, task2);
```

with the following lines in the previous program:

```
task1.Wait();
task2.Wait();
```

**E2.7** If you execute the following code, can you predict the output?

```
using static System.Console;
WriteLine("Exercise 2.7");

try
{
    var task1 = Task.Run(() => throw new
     InvalidOperationException("invalid operation"));
    var task2 = Task.Run(() => throw new
     OutOfMemoryException("insufficient  memory"));
```

```
    Task.WaitAny(task1, task2);
    WriteLine("End");
}
catch (AggregateException ae)
{
    ae.Handle(e =>
    {
        if (e is InvalidOperationException |
            e is OutOfMemoryException
        )
        {
            WriteLine($"Caught error: {e.Message}");
            return true;
        }
        return false;
    }
    );
}
```

**E2.8** If you execute the following code, can you predict the output?

```
using static System.Console;
WriteLine("Exercise 2.8");

try
{
    var someTask = Task.Run(() => throw new
     InvalidOperationException("invalid operation")
     { Source = "Task1"});
    // Allowing some time so that task can get up and running
    while (someTask.Status != TaskStatus.Running)
    {
        Thread.Sleep(10);
    }
```

```
    // Waiting for the state change now
    while (someTask.Status == TaskStatus.Running)
    {
        Thread.Sleep(10);
    }
    WriteLine($"SomeTask's status: {someTask.Status}");
    WriteLine("The application ends here.");
}
catch (AggregateException ae)
{
    ae.Handle(e =>
    {
        if (e is InvalidOperationException )
        {
            WriteLine($"Caught error: {e.Message} Source=
             {e.Source}");
            return true;
        }
        return false;
    }
    );
}
```

**E2.9** If you execute the following code, can you predict the output?

```
using static System.Console;

var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
var printTask = Task.Run
 (
  () =>
```

```
  {
      int i = 0;
      while (i != 10)
      {
          if (token.IsCancellationRequested)
          {
              WriteLine("Cancelling the print activity.");
              return;
          }
          // Do some work, if required.
          Thread.Sleep(1000);
          i++;
      }
  }, token
);

Thread.Sleep(500);
WriteLine("Task cancellation initiated.");
tokenSource.Cancel();
// Wait till the task finishes the execution
while (!printTask.IsCompleted) { }
WriteLine($"The final status of printTask is: {printTask.Status}");
WriteLine("End of the main thread.");
```

**E2.10** In the previous exercise, if you replace the following code segment:

```
  if (token.IsCancellationRequested)
  {
      WriteLine("Cancelling the print activity.");
      return;
  }
```

with the following line:

```
token.ThrowIfCancellationRequested();
```

will be there any change in the output?

**E2.11** The following program creates a parent task and a nested task. It also allows you to cancel the nested task if you enter C quickly. Check whether you can predict the output.

```
using static System.Console;

WriteLine("Exercise 2.11");
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
var parent = Task.Factory.StartNew(
    () =>
    {
        var finalResult = string.Empty;
        try
        {
            if (token.IsCancellationRequested)
            {
               finalResult = "The cancellation request is
                raised before the start of the parent task.";
               token.ThrowIfCancellationRequested();
            }
            WriteLine("The parent task has started.");

            // Creating a nested task
            var child = Task.Factory.StartNew(
                () =>
                {
                    WriteLine("The nested task has started.");
```

```
                    for (int i = 0; i < 10; i++)
                    {
                        token.ThrowIfCancellationRequested();
                        WriteLine($"\tThe nested task
                         prints:{i}");
                        Thread.Sleep(100);
                    }
                    return "The nested task has finished too.";
                }, token);

            finalResult = "Parent task: completed. Additional
             info: n/a.";

            // Updating the final result with the result of the
            // nested task
            finalResult = $"Parent task: completed.\t
             Additional info: {child.Result} ";

        }
        catch (AggregateException ae)
        {
            foreach (Exception e in ae.InnerExceptions)
            {
                WriteLine($"Caught error: {e.Message}");
            }
        }
        catch (OperationCanceledException oce)
        {
            WriteLine($"Error: {oce.Message}");
        }
        return finalResult;
    });
```

```
WriteLine("Enter c to cancel the nested task.");
char ch = ReadKey().KeyChar;
if (ch.Equals('c'))
{
    WriteLine("\nTask cancellation requested.");
    tokenSource.Cancel();
}

WriteLine($"Status: {parent.Result}");
WriteLine("End of the main thread.");
```

**E2.12** If you execute the following code, can you predict the output?

```
using static System.Console;
TaskCompletionSource<int> tcs = new();
int value = 10;
var task1=Task.Run(() => value++);
Thread.Sleep(1000);
var task2=Task.Run(() =>
{
    Thread.Sleep(2000);
    tcs.SetResult(value*10);
}
);
Thread.Sleep(1000);
WriteLine($"The final result is:{tcs.Task.Result}");
```

**E2.13 In Chapter 1 you solved exercise 1.6. Since you learned about implementing exception and cancellation scenarios, can you solve that exercise considering these scenarios?**

# Summary

This chapter continued the discussion of task programming, but this time the focus was on handling various special scenarios with different examples and case studies. In brief, it answered the following questions:

- How do you handle exceptions in a parallel programming environment?
- How can you support cancellations in a parallel programming environment?
- How can you monitor cancellations in your applications?
- How can you manage some I/O-bound tasks using the `TaskCompletionSource` class?

# Solutions to Exercises

Here is a sample solution set for the exercises in this chapter.

## E2.1

This program produces the following output:

```
Exercise 2.1
End
```

**Additional Note:** You do not observe the exception because the main thread did not encounter the exception; it was encountered by the task that was created by this main thread. I covered this topic using Demonstration 2 and Demonstration 3.

## E2.2

This time you'll see the exception name. You may note that instead of seeing `System.DivideByZeroException`, you'll see `System.AggregateException` as follows:

```
Exercise 2.2
Encountered with System.AggregateException
End
```

## E2.3

This program produces the following output:

```
Exercise 2.3
Caught error: invalid data
Caught error: insufficient memory
```

## E2.4

The call to the statement `task1.Wait();` causes the `InvalidDataException`. As a result, control leaves the `try` block and produces the following output:

```
Exercise 2.4
Caught error: invalid data
```

## E2.5

The program produces the following output:

```
Exercise 2.5 and Exercise 2.6
Caught inside DoSomething: invalid data
Caught inside main: insufficient memory
```

# E2.6

The program produces the following output (see the explanation of E2.4 if required):

```
Exercise 2.5 and Exercise 2.6
Caught inside DoSomething: invalid data
```

# E2.7

This program produces the following output:

```
Exercise 2.7
End
```

You may be wondering why are you not seeing the task's exception(s) in the output. This is because when you use the `WaitAny` method, the task's exception does not propagate to the `AggregateException`. I encourage you to read Stephen Clearly's blog post at https://blog.stephencleary.com/2014/10/a-tour-of-task-part-5-wait.html that summarizes the difference among `WaitAny`, `Wait`, and `WaitAll` as follows:

> *The semantics of `WaitAny` are a bit different than `WaitAll` and `Wait`: `WaitAny` merely waits for the first task to complete. It will not propagate that task's exception in an `AggregateException`. Rather, any task failures will need to be checked after `WaitAny` returns. `WaitAny` will return -1 on timeout, and will throw OperationCanceledException if the wait is cancelled.*

# E2.8

This program may produce the following output:

```
Exercise 2.8
SomeTask's status: Faulted
The application ends here.
```

**However, you may also notice that the program is stuck executing**. Why? Notice that when you evaluate the following condition, the task may not be in the `Running` state (for example, it may be in the `WaitingForActivation` state):

```
// Allowing some time so that the task can get up and running
while (someTask.Status != TaskStatus.Running)
{
  Thread.Sleep(10);
}
```

But inside the loop when you invoke the `Sleep` method, by that time, the task may transition from the `Running` state to one of the terminating states. As a result, the program will stuck here.

This is why to get a consistent output, you'd like to replace the following segment:

```
// Allowing some time so that task can get up
// and running( Program can stuck here)
 while (someTask.Status != TaskStatus.Running)
 {
     Thread.Sleep(10);
 }

 // Waiting for the state change now
 while (someTask.Status == TaskStatus.Running)
 {
     Thread.Sleep(10);
 }
```

with the following one: `while (!someTask.IsCompleted) { }`

**Note** I already mentioned that you should avoid this kind of polling in the production code as it is inefficient.

## E2.9

Here is some possible output:

```
Task cancellation initiated.
Cancelling the print activity.
The final status of printTask is: RanToCompletion
End of the main thread.
```

## E2.10

This time the final task status should appear as `Canceled`. Here is some sample output:

```
Task cancellation initiated.
The final status of printTask is: Canceled
End of the main thread.
```

## E2.11

You already know that this program creates a parent task and a nested task. It also allows you to cancel the nested task if you enter C quickly. As a result, you may see different output. For example, if you do not intend to cancel the nested task and press the Enter key at the end, you can see the following output:

```
Exercise 2.11
Enter c to cancel the nested task.
The parent task has started.
```

```
The nested task has started.
        The nested task prints:0
        The nested task prints:1
        The nested task prints:2
        The nested task prints:3
        The nested task prints:4
        The nested task prints:5
        The nested task prints:6
        The nested task prints:7
        The nested task prints:8
        The nested task prints:9
Status: Parent task: completed.  Additional info: The nested
task has finished too.
End of the main thread.
```

On the other hand, depending on the time of cancellation, you can get different outputs. For example, if you press C almost at the beginning of the application, you will see the following:

```
Exercise 2.11
Enter c to cancel the nested task.
c
Task cancellation requested.
Error: The operation was canceled.
Status: The cancellation request is raised before the start of
the parent task.
End of the main thread.
```

Otherwise, you may see something like the following:

```
Exercise 2.11
Enter c to cancel the nested task.
The parent task has started.
```

```
The nested task has started.
        The nested task prints:0
        The nested task prints:1
        The nested task prints:2
        The nested task prints:3
c
Task cancellation requested.
Caught error: A task was canceled.
Status: Parent task: completed. Additional info: n/a.
End of the main thread.
```

## E2.12

You should see the following output:

```
The final result is: 110
```

[**Clue**: Notice that `task1` updates the initial value to 11, but `task2` further sets it as 11*10=110. The sleep statements are placed to preserve the order of evaluation.]

## E2.13

I leave this exercise to you now. Good luck!

CHAPTER 3

# Exploring Synchronization and Concurrent Collections

In a multithreaded environment, managing the shared data is a challenging activity. This is because multiple threads may try to perform the reading and writing operations simultaneously. If you do not coordinate them, you may see erroneous outcomes. Synchronization mechanisms help you control access to shared data to avoid inconsistent or erroneous outcomes.

This chapter discusses some useful synchronization techniques. It also covers the usage of concurrent collections in our applications. These concepts will help you make applications that can work seamlessly in a multithreaded environment. Since parallel programming is a subset of multithreading, understanding these concepts will help you a lot.

## Synchronization

Synchronization is a vast topic, and there are many useful constructs. Depending on the situation, you may prefer one construct over another. One of my favorite authors is Joseph Albahari (the author of *C# in a*

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5_3

*Nutshell*). He categorized the synchronization constructs as follows (see https://www.albahari.com/threading/part2.aspx#_Synchronization):

- **Simple blocking methods:** You already saw these methods. For example, `Thread.Sleep` and `Task.Wait` belong to this category.
- **Locking constructs:** Some examples in this category include the `lock` statements of `Mutex`, `Semaphore`, `SemaphoreSlim`, and so on. These are further categorized by exclusive locking and nonexclusive locking constructs. For example, `lock` and `Mutex` fall into exclusive locking constructs, whereas `Semaphore` and `SemaphoreSlim` fall into the other category.
- **Signaling constructs:** Event wait handles such as `AutoResetEvent`, `ManualResetEvent`, and `CountDownEvent` belong to this category.
- **Nonblocking synchronization constructs**: The `Interlocked` class and the `volatile` keyword are common examples in this category.

**POINT TO NOTE**

Discussing all of these constructs would make this chapter unnecessarily fat. In fact, to understand the content of the rest of the book and do the exercises, you may need to understand only a few of them. **So, I decided to cover at least one from each category.** However, if interested, you can refer to Appendix A where you'll find some additional material. Since you already saw me using `Thread.Sleep` and `Task.Wait`, let's focus on the remaining categories in this chapter.

# Understanding Why We Need Synchronization

Before I discuss the synchronization primitives, I will present an erroneous program that shows why we need synchronization techniques.

## An Erroneous Program

The following program tries to mimic a typical banking scenario where you see five different tasks perform some common operations: three of them perform the credit operations, and two of them perform the debit operations. Let's assume the credit amounts are $100, $200, and $300, whereas the debit amounts are $500 and $100. If you run these tasks one by one, the initial balance and the final balance will be the same. For example, if the initial balance is $0, after performing these operations, the final balance will also be $0. However, is this the case when these tasks run in a multithreaded environment?

## Demonstration 1

Let's run the following program in a multithreaded environment and see the output:

```
using static System.Console;

Account account = new();
var credit100 = Task.Run(() => account.Credit(100));
var debit500 = Task.Run(() => account.Debit(500));
var credit200 = Task.Run(() => account.Credit(200));
var credit300 = Task.Run(() => account.Credit(300));
var debit100 = Task.Run(() => account.Debit(100));

Task.WaitAll(credit100, credit200, credit300, debit500,
 debit100);
WriteLine($"The current balance is ${account.Balance}");
```

```
#region Non-synchronized version
class Account
{
   public decimal Balance { get; set; }
   public void Credit(decimal amount)
   {
    Balance += amount;
    WriteLine($"The balance after the credit of ${amount} is
     ${Balance}");
   }
   public void Debit(decimal amount)
   {
    Balance -= amount;
    WriteLine($"The balance after the debit of ${amount} is
     ${Balance}");
   }
}
#endregion
```

## Output

You may see different outputs on different executions. However, when I executed this program, most of the time, I got incorrect results. And on a few occasions, I saw that the final balance was $0; in those cases, also the intermediate results were wrong. Let me show you a sample:

```
The balance after the credit of $200 is $-500
The balance after the debit of $500 is $-500
The balance after the debit of $100 is $-300
The balance after the credit of $300 is $-200
The balance after the credit of $100 is $-500
The current balance is $-300
```

## Analysis

Notice that multiple tasks were working on nonatomic operations that were not synchronized. This allowed multiple threads to enter the critical section of the code, and as a result, you see this incorrect result.

## Q&A Session

**Q3.1 How do the atomic operations work in a multithreaded environment?**

Consider this line of code: `Balance += amount;`. What does it do? Surely, it increments the `Balance` by the `amount`. However, it may not be a single operation; instead, multiple steps can be involved as follows:

- **Step 1:** Load the current value of the `Balance` into a register.
- **Step 2**: Increment this current value by the amount (you may store the result in a temporary variable as well).
- **Step 3:** Store/set the updated value back to the `Balance`.

Now consider the case when a thread, say, thread A, starts executing these steps. Let's assume that it executes step 1 and step 2, but before executing step 3, it is preempted by another thread, say, thread B. Let's also assume that thread B completes all the steps without any interference from other threads. Now thread A resumes execution and overwrites the final value of `Balance`. This means the change made by thread B is lost.

**Q3.2 What is a critical section?**

In simple words, a critical section is a piece of code that you'd like to guard in a multithreaded environment. In this region/section, you coordinate access to the shared variables to produce the correct result. In

a moment, I'll use the `lock` statement to make a critical section around my code. Let me show you a sample from the upcoming demonstration:

```
public void Credit(decimal amount)
{
    lock (_balanceLock)
    {
       Balance += amount;
       WriteLine($"The balance after credit ${amount} is
        {Balance} $");
    }
}
```

Here the `lock` keyword plays the role of a gatekeeper who allows only one thread to work in the critical section.

## Using Lock Statements

Lock statements are usually used when we want exclusive access to a shared resource. The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock).

> *The lock statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released. The lock statement ensures that at maximum only one thread executes its body at any time moment.*

## Demonstration 2

The previous quote is self-explanatory. It ensures that while a thread is working in a critical section, no other thread can interfere with it. So, let's modify the Account class as follows:

```
#region Synchronized version using the lock statement
class Account
{
  public decimal Balance { get; set; }
  private readonly object _balanceLock = new();
  public void Credit(decimal amount)
  {
   lock (_balanceLock)
    {
    Balance += amount;
    WriteLine($"The balance after the credit of ${amount} is
     ${Balance}");
    }
   }
   public void Debit(decimal amount)
   {
    lock (_balanceLock)
    {
     Balance -= amount;
     WriteLine($"The balance after the debit of ${amount} is
      ${Balance}");
    }
   }
}
#endregion
```

**Note** To avoid repetition, I did not show the complete program. However, you can download the project `Demo2_UsingLock` to run the complete program.

## Output

Run this modified program again. Here is some sample output:

```
The balance after the credit of $100 is $100
The balance after the credit of $200 is $300
The balance after the debit of $500 is $-200
The balance after the credit of $300 is $100
The balance after the debit of $100 is $0
The current balance is $0
```

You may see different intermediate results, but each time you will see the final balance is $0 for sure.

# Using Interlocked Classes

Using the `lock` statements, I solved the unwanted entries in the output. This is a common approach. However, you can adopt a different approach as well. To illustrate, this time I'll show you how to use the `Interlocked` class that provides atomic operations for variables that can be shared by multiple threads. By using this, you can enjoy **lock-free programming**.

`Interlocked` is a static class that has some useful methods such as `Add`, `Increment`, `Decrement`, `Exchange`, `CompareExchange`, and so on. You'll also see many overloaded versions. Figure 3-1 shows four overloads for the `Add` method.

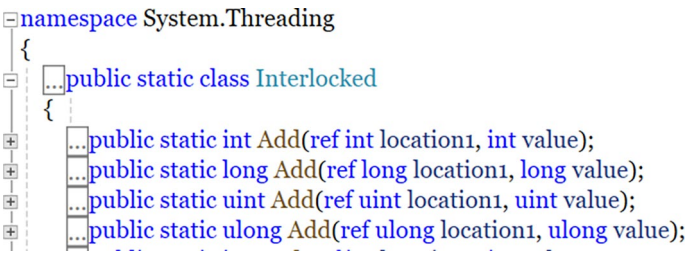

namespace System.Threading
{
public static class Interlocked
{
public static int Add(ref int location1, int value);
public static long Add(ref long location1, long value);
public static uint Add(ref uint location1, uint value);
public static ulong Add(ref ulong location1, ulong value);

***Figure 3-1.*** *The overloaded Add methods in the Interlocked class*

Let me expand a method description:

```
//
// Summary:
//     Adds two 32-bit integers and replaces the first
// integer with the sum, as an atomic operation.
//
// Parameters:
//   location1:
//     A variable containing the first value to be added.
// The sum of the two values is stored in location1.
//
//   value:
//     The value to be added to the integer at location1.
//
// Returns:
//     The new value that was stored at location1 by this
// operation.
//
// Exceptions:
//   T:System.NullReferenceException:
//     The address of location1 is a null pointer.
public static int Add(ref int location1, int value);
```

You can see that this method can add two integers atomically. However, notice the presence of the `ref` parameter. Therefore, if you write something like this in the `Account` class:

```
public void Credit(int amount)
 {
     Interlocked.Add(ref Balance, amount);
 }
```

you'll see the compile-time error:

```
CS0206 A non ref-returning property or indexer may not be used
as an out or ref value
```

Now you understand that if you want to use the `Add` method, you need to make some changes to the code.

## Demonstration 3

Let's modify the `Account` class and introduce a backing field inside it as follows:

```
// The New Look of the Account Class
class Account
{
    // public decimal Balance { get; set; }
    private int _balance;
    public int Balance
    {
        get { return _balance; }
        set { _balance = value; }
    }

    public void Credit(int amount)
    {
        Interlocked.Add(ref _balance, amount);
    }
```

```
    public void Debit(int amount)
    {
        Interlocked.Add(ref _balance, -amount);
    }
}
```

---

**Note** To avoid repetition, I did not show the complete program. However, you can download the project `Demo3_InterlockedOps` to run the complete program.

---

## Output

Run this modified program again. You will always see the following output:

```
The current balance is $0
```

You can see that this approach will produce the correct output consistently.

## Q&A Session

**Q3.3 You showed some task continuation examples in Chapter 1. By using that mechanism, can I avoid locking?**

Yes, task continuations can indeed remove the necessity of using a lock or providing signals in certain scenarios. However, there are many situations where you'll reap benefits by using other synchronization constructs. The demonstrations that used the `lock` statements and the `Interlocked` class were good examples in this context. If required, you can combine different techniques in an application as well.

## Signaling Using AutoResetEvent

Consider a multithreading environment where several threads try to access a shared resource. Now assume that any of those threads get access to this resource. Once the demand is fulfilled, this thread notifies others by saying that they can use this resource now. This kind of notification is known as **signaling**. This section discusses one signaling construct to demonstrate how different tasks (more accurately, threads) can coordinate their executions.

You are already familiar with locking mechanisms where, by releasing a lock, a thread notifies another thread that it can proceed. It is also one kind of signaling. However, this section describes a different construct (event wait handles) that uses a different kind of signaling. Let's see this.

**POINT TO NOTE**

I already told you that synchronization is a vast topic. To give you an idea of signaling using event wait handles, this chapter discusses the `AutoResetEvent` class only. See Appendix A for a discussion of the `ManualResetEvent` and the `CountdownEvent` classes.

In my city, if a passenger likes to use the metro railway service, he needs to purchase a ticket that comes in the form of a token. (Frequent travelers can purchase a card and top off the card as well.) Metro railway premises are highly secured. Passengers form a queue and enter the restricted area one by one by scanning their tokens. It is obvious that you need to form (or maintain) the queue if multiple passengers are waiting to enter the area at the same time; otherwise, you can disrupt the flow and cause chaos. **However, there is another characteristic in this scenario: the gate automatically closes once a passenger steps through. It is similar to a**

**ticket turnstile in which by inserting a ticket only one passenger can pass through and then the gate closes automatically.** `AutoResetEvent` perfectly mimics these scenarios. The documentation defines the `AutoResetEvent` class as follows (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.autoresetevent?view=net-8.0):

> *Represents a thread synchronization event that, when signaled, releases one single waiting thread, and the event resets automatically. If no thread is waiting, the next thread that is set to the waiting state is immediately released, and the event resets automatically.*

Now the question arises: how to use `AutoResetEvent`? It is simple: you need to consider the following methods:

- **WaitOne**: A thread calls this method and enters the waiting (aka blocking) state.
- **Set**: An unblocked thread (that can access the `AutoResetEvent` instance) calls this method on it to release one blocked thread.

Do not worry! The upcoming example will make their usage clearer to you. Before I demonstrate the example, **let's see how to create an `AutoResetEvent` Instance.**

From the Visual Studio IDE, you can retrieve the following:

```
namespace System.Threading
{
    public sealed class AutoResetEvent : EventWaitHandle
    {
        public AutoResetEvent(bool initialState) :
         base(initialState,
         EventResetMode.AutoReset) { }
    }
}
```

This indicates you can create an AutoResetEvent instance as follows:

```
AutoResetEvent autoResetEvent = new(false);
```

Since the AutoResetEvent class derives from the EventWaitHandle class, you can write an equivalent code as follows:

```
EventWaitHandle autoResetEvent = new AutoResetEvent(false);
```

Interestingly, the EventWaitHandle class has several constructors, and each of them accepts EventResetMode as a parameter. Here is the simplest one:

```
public EventWaitHandle(bool initialState, EventResetMode mode);
```

This construct allows you to write another equivalent code as follows:

```
EventWaitHandle autoResetEvent = new (false,EventResetMode
  .AutoReset);
```

Finally, I assume I do not need to remind you that you can use the var keyword as well. For example, the following line of code is also valid:

```
var autoResetEvent = new AutoResetEvent(false);
```

---

**Note** You may notice that while making an instance of the AutoResetEvent class, I passed false into the constructor. If you pass true into the constructor, the Set method on it is called immediately. In other words, you start with the signaled state, and a thread is released immediately. However, in my examples, I'll opt to pass false. This is because I have not seen any metro station where the officials allow the first customer to enter the restricted area without checking.

---

## Demonstration 4

Let's consider a scenario where three potential customers want to enter a shopping mall (you can consider the metro railway station as well). *Let's assume that two customers have reached the entry gate almost at the same time. Shortly thereafter, the third customer also reaches the entry gate of this shopping mall.*

Either customer 1 or customer 2 will enter the mall first, and customer 3 should be the last one to enter. Let's see a program that mimics this scenario using `AutoResetEvent`.

```
using static System.Console;

var resetEvent = new AutoResetEvent(false);

WriteLine("Two customers are approaching  the mall.");
Task.Run(VisitMall);
Task.Run(VisitMall);

// Imposing some delay to mimic a real-world scenario
Thread.Sleep(2000);

WriteLine("Press any key to issue the signal from the main thread.");
ReadKey();
resetEvent.Set();
Thread.Sleep(1000);

// Reset is not required to close the gate

WriteLine("Press any key to issue another signal from the main
 thread.");
ReadKey();
resetEvent.Set();
Thread.Sleep(1000);
```

```
WriteLine("Another customer is approaching the mall.");
Task.Run(VisitMall);

WriteLine("Press any key to exit.");
ReadKey();

void VisitMall()
{
    // Imposing a small delay to mimic a real-world scenario
    Thread.Sleep(1000);
    WriteLine($"The customer {Task.CurrentId} is waiting for the
     entry pass.");
    resetEvent.WaitOne();
    WriteLine($"The customer {Task.CurrentId} enters the mall.");
}
```

## Output

Here is some sample output (I pressed the Enter key to issue the signal):

```
Two customers are approaching the mall.
The customer 1 is waiting for the entry pass.
The customer 2 is waiting for the entry pass.
Press any key to issue the signal from the main thread.
The customer 1 enters the mall.
Press any key to issue another signal from the main thread.
The customer 2 enters the mall.
Another customer is approaching the mall.
Press any key to exit.
The customer 3 is waiting for the entry pass.
```

## Analysis

This output reflects the following characteristics:

- This program essentially demonstrated one-way signaling where the worker threads were waiting for a signal from the main thread.

---

**Note** You may note that the reverse flow is also possible. While solving the exercises, you will see a sample implementation (see the solution of E3.6) that demonstrates two-way signaling as well.

---

- Initially, both customer 1 and customer 2 were waiting for the entry pass, but once the main thread issued the signal, only customer 1 got the pass! **This depicts the fact that a signal could unblock only one waiting task (aka thread)**.
- **Notice that the gate was closed automatically.** This is why no other customer could enter the mall automatically after customer 1 entered the mall. The second customer could enter the mall only when the main thread issued another signal.
- Now it makes sense why the third customer could not enter the mall yet! This customer can enter if and only if the main thread issues another signal.

**Author's Note:** I want you to note the last two points. While using `ManualResetEvent`, you'll notice a significant difference in this area. Let me show you some sample output. When I replaced the following line

`var resetEvent = new AutoResetEvent(false);` with `var resetEvent = new ManualResetEvent(false);` , I got the following sample output (I pressed the Enter key to issue the signal):

```
Two customers are approaching the mall.
The customer 1 is waiting for the entry pass.
The customer 2 is waiting for the entry pass.
Press any key to issue the signal from the main thread.
The customer 1 enters the mall.
The customer 2 enters the mall.
Press any key to issue another signal from the main thread.
Another customer is approaching the mall.
Press any key to exit.
The customer 3 is waiting for the entry pass.
The customer 3 enters the mall.
```

Have you noticed the difference? Both customer 1 and customer 2 were able to enter the mall when the main thread issued the signal. In addition, customer 3 was also able to enter the mall because in between, the gate was not closed by calling the `Reset` method. However, do not worry! You'll see more discussion of them in Appendix A.

## Q&A Session

**Q3.4 It appears to me that every call to the `Set` method releases a thread. Is this correct?**

Good question. Most of the time the answer is yes; however, there can be exceptional situations. In this context, Microsoft states the following (see the previously mentioned link):

> *There is no guarantee that every call to the Set method will release a thread. If two calls are too close together, so that the second call occurs before a thread has been released, only one thread is released. It's as if the second call did not happen.*

**Q3.5 Will it be a problem when a thread calls the `Set` method when no other threads are waiting?**

It should not cause any typical problems for you. In this case, the handle remains open until another thread calls `WaitOne`.

**Q3.6 "In this case, the handle remains open until another thread calls `WaitOne`." Is there any specific thought behind this design choice?**

Consider a scenario where two threads are participating in a multithreading environment. One thread calls the `Set` method, and another thread calls the `WaitOne` method simultaneously. However, assume that the `Set` method is executed just before the `Waitone` method. Let's assume that since there is no one waiting, you closed the gate now. However, in between, another thread invoked the `WaitOne` method; so it keeps waiting indefinitely. This is like a situation when you are waiting for a gate to be opened, but there is no queue. Now you understand that the proposed design prevents the race condition between threads.

**Q3.7 I assume a situation where multiple threads will wait for the signals (and my code uses `AutoResetEvent`). Is it possible for me to call the `Set` method multiple times in advance to release them immediately?**

When I showed you a sample output by using `ManualResetEvent`, you saw that this is possible. However, if your code uses `AutoResetEvent`, you need to remember the following.

If the `Set` method is called when there is no blocking thread, the handle stays open; however, it does not allow you to make multiple signals in advance. So, it does not matter how many times you release a signal in advance; when those threads appear, only one of them will be released immediately. As a result, the other threads will keep waiting for the signals.

**Author's Note:** Exercise 3.3 is made to demonstrate this scenario.

# Concurrent Collections

Concurrent collections are an essential part of parallel programming. In a multithreaded environment, they fulfill the need for thread-safe collections. In fact, they are optimized for high-concurrency scenarios. Let's start discussing them.

## System.Collections.Concurrent Namespace

I assume you are familiar with standard generic collections where you saw several types in the `System.Collections.Generic` namespace. However, concurrent collections are different from those ordinary collections. Here we'll focus on the `System.Collections.Concurrent` namespace that contains a limited number of types. In this section, I'll start the discussion of the `IProducerConsumerCollection<T>` interface. Later, I'll discuss the following types:

- `ConcurrentStack<T>`
- `ConcurrentQueue<T>`
- `ConcurrentDictionary<TKey,TValue>`
- `ConcurrentBag<T>`
- `BlockingCollection<T>`

You can safely assume that `ConcurrentStack<T>`, `ConcurrentQueue<T>`, and `ConcurrentDictionary<TKey,TValue>` are made for their nonconcurrent versions: `Stack<T>`, `Queue<T>`, and `Dictionary<TKey,TValue>`.

However, you can see that there are two additional concurrent classes called `ConcurrentBag<T>` and `BlockingCollection<T>`. You'll not find any nonconcurrent equivalent for them.

**POINT TO NOTE**

The `System.Collections.Concurrent` namespace contains a few more types. I believe that a quick discussion on the mentioned types will give you a fair idea about the usage of the concurrent collections. If interested, you can learn about the remaining types from https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent?view=net-8.0.

On the contrary, some classes do not have a counterpart in the concurrent library. The `List<T>`, `SortedList<TKey,TValue>`, and `SortedDictionary<TKey,TValue>` are a few such examples.

## IProducerConsumerCollection<T> Interface

This interface defines methods to work with thread-safe collections that are intended for producer/consumer usage. In a producer/consumer collection, a producer produces some items (for example, add items), and a consumer consumes those items (for example, retrieves items while removing). If you are familiar with stacks and queues, you already know this.

The following points that summarize some of the important characteristics of the producer-consumer pattern:

- Developers use this pattern to decouple the process that produces items from the process that consumes those items.
- The production rate and consumption rate can be different.
- Normally, we use a queue to store the items. This is because the item that is produced first is usually consumed first as well.

The `ConcurrentQueue<T>`, `ConcurrentStack<T>`, and `ConcurrentBag<T>` classes implement the `IProducerConsumerCollection<T>` interface. If you investigate in Visual Studio, you'll see the screen in Figure 3-2.

```
namespace System.Collections.Concurrent
{
    public interface IProducerConsumerCollection<T> : IEnumerable<T>, IEnumerable, ICollection
    {
        void CopyTo(T[] array, int index);
        T[] ToArray();
        bool TryAdd(T item);
        bool TryTake([MaybeNullWhen(false)] out T item);
    }
}
```

***Figure 3-2.*** *The IProducerConsumerCollection<T> interface inside the System.Collections.Concurrent namespace*

You can see that `IProducerConsumerCollection<T>` inherits from three more interfaces and contains four methods: `CopyTo`, `ToArray`, `TryAdd`, and `TryTake`. The last two will be useful for our upcoming discussions. Let's learn about them.

- The `TryAdd` method attempts to add an object to `IProducerConsumerCollection<T>`. If the object is added, the method returns `true`; otherwise, it returns `false`.
- The `TryTake` method attempts to remove and return an object from `IProducerConsumerCollection<T>`. The method returns `true` if the object is removed and returned successfully. Otherwise, it will return `false`. (You may note that the `item` parameter in the method signature is used to contain the removed object.)

You know that if a concrete class implements the `IProducerConsumerCollection<T>` interface, it must provide the implementations of the members in the interface. (Starting from C# 8.0, you can have default interface methods that can have some default implementation. However,

I am not talking about them here.) Since the ConcurrentQueue<T>, ConcurrentStack<T>, and ConcurrentBag<T> classes implement the IProducerConsumerCollection<T> interface, they must provide the implementation for the TryAdd and TryTake methods.

Let's see how the ConcurrentStack<T> class implements the TryAdd method. From Visual Studio, you can see the following explicit implementation:

```
bool IProducerConsumerCollection<T>.TryAdd(T item)
{
    Push(item);
    return true;
}
```

where the Push method is further defined as follows:

```
public void Push(T item)
{
    Node newNode = new Node(item);
    newNode._next = _head;
    if (Interlocked.CompareExchange(ref _head, newNode,
      newNode._next) != newNode._next)
    {
    PushCore(newNode, newNode);
}
```

You can see that in the case of a stack, the TryAdd method internally calls the Push method. The presence of the Interlocked operation in the Push method ensures that while using this method, you do not need to worry about the lock statements.

Similarly, if you investigate how the ConcurrentQueue<T> class implements the TryAdd method, you'll see the following:

```
bool IProducerConsumerCollection<T>.TryAdd(T item)
 {
     Enqueue(item);
     return true;
 }
```

You can see that in case of a queue, the TryAdd method internally calls the Enqueue method. On further study, you'll see that the Enqueue method is defined as follows:

```
    public void Enqueue(T item)
    {
        if (!_tail.TryEnqueue(item))
        {
            EnqueueSlow(item);
        }
     }
```

However, the EnqueueSlow method uses the lock statement as follows:

```
  private void EnqueueSlow(T item)
  {
      while (true)
      {
          ConcurrentQueueSegment<T> tail = _tail;

          // Try to append to the existing tail.
          if (tail.TryEnqueue(item))
          {
             break;
          }
```

```
lock (_crossSegmentLock)
{
// The remaining code is not shown
```

**POINT TO NOTE**

From these code segments, you can see that the `ConcurrentStack<T>` and `ConcurrentQueue<T>` classes implement the `TryAdd` method explicitly and expose the same functionality using specifically named public methods such as `Push` and `Enqueue`. You may also note that I have picked these code segments while executing my programs in .NET 8. However, if you investigate the same code in .NET 7, you'll see many supporting comments that can help you understand these functions better.

Have you seen any other important observations from these code segments? If not, do not worry! You'll see them in the next section when I'll summarize some other important points as well. For now, let's talk about the `TryTake` method.

In Figure 3-2, I showed you the `IProducerConsumerCollection<T>` interface details from the Visual Studio IDE. This figure showed that the `TryTake` method has the following signature:

```
bool TryTake([MaybeNullWhen(false)] out T item);
```

You understand that an implementer of `IProducerConsumerCollection<T>` must provide the implementation for this method. I already investigated the `TryAdd` method in the `ConcurrentStack<T>` and `ConcurrentQueue<T>` classes. You can repeat a similar investigation and essentially learn the following:

- In the case of a stack, the `TryTake` method removes the most recently added element.

- In the case of a queue, the `TryTake` method removes the least recently added element.
- In the case of a bag, the `TryTake` method removes whatever element it can remove most efficiently.

## Notable Characteristics

From our discussion on the concurrent collections so far, let me point out the notable characteristics. I suggest you read them carefully before you jump into the next section.

1. **When you see a concurrent collection for its noncurrent equivalent, you can find similarities.** It is obvious that there are some differences in the access mechanisms, but the general concept is the same. For example, both `ConcurrentQueue<T>` and `Queue<T>` support the first-in, first-out mechanism. So, if you are familiar with the `Queue<T>`, you can easily understand the `ConcurrentQueue<T>`.
2. **Consumers of these collections do not need to worry about the locks.** You may find that the necessary lock statements are provided internally in these structures. For example, when I showed you the `TryAdd` implementations in `ConcurrentStack<T>` and `ConcurrentQueue<T>`, you saw that the designers already used `lock` and `interlocked` internally in the respective places. This gives you the clue that Microsoft made concurrent collections by using the lock statements internally to make them thread-safe. In fact, while discussing concurrent dictionaries, Microsoft also ensures the

following (see https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2?view=net-8.0)

*For modifications and write operations to the dictionary, ConcurrentDictionary<TKey,TValue> uses fine-grained locking to ensure thread safety.*

**POINT TO NOTE**

In this context, I also encourage you to read the Microsoft blog at https://devblogs.microsoft.com/pfxteam/faq-are-all-of-the-new-concurrent-collections-lock-free/ to get more information. However, I suggest you read this blog once you are familiar with the common operations of these concurrent collections and acquire some basic ideas about memory barriers.

3. In the built-in implementations (that I showed you when I investigated the `TryAdd` method in `ConcurrentStack<T>` and `ConcurrentQueue<T>`), **you can also see how to use linked lists, which are conducive to lock-free or low-lock implementations**.

4. You saw that the `ConcurrentStack<T>` and `ConcurrentQueue<T>` classes implement the `TryAdd` method explicitly and expose the same functionality using specifically named public methods such as `Push` and `Enqueue`. **If you investigate the `TryTake` method, you'll have a similar observation. The only difference is that this time you'll see the methods `TryPop` (for the stack) and `TryDequeue` (for the queue).**

5. Shortly, you'll see the `ConcurrentDictionary<TKey, TValue>` class. It is the only concurrent collection that provides access to a specific item using a key. For example, if a dictionary contains 10 elements, you can get the fifth element (without removing it) using the key.

**POINT TO NOTE**

You may note that the `ConcurrentStack<T>`, `ConcurrentQueue<T>`, and `ConcurrentBag<T>` classes also have a `TryPeek` method that you can use to see the next item in the collection without removing it. However, you may note that the `TryPeek` method in these classes lets us look only one item deep into the collection, whereas the `ConcurrentDictionary<Tkey,TValue>` class lets you see any item using the key.

6. About the `IProducerConsumerCollection<T>` interface, the official documentation states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.iproducerconsumercollection-1?view=net-8.0):

*Defines methods to manipulate thread-safe collections intended for producer/consumer usage. This interface provides a unified representation for producer/consumer collections so that higher level abstractions such as BlockingCollection<T> can use the collection as the underlying storage mechanism.*

It is obvious that the `IProducerConsumerCollection<T>` types can be used based on your needs. In addition, this official documentation also states that **they can serve as backing storage(s) when you use them with the `BlockingCollection<T>` class.** (You will learn more about this when I discuss the `BlockingCollection<T>` class.)

## Q&A Session

**Q3.8 Why do we need concurrent collections when we can use common synchronization techniques such as lock statements to ensure thread safety?**

Using a lock statement, you essentially turn a multithreaded code segment into a single-threaded code segment (at least within the lock statement). As a result, you essentially defeat the associated benefits of using a multithreaded application.

**Q3.9 Does the use of a few lock statements matter for a simple application?**

It is true that in the case of a tiny application, using a few lock statements does not matter much. However, what about an enterprise application? If you need to use the `lock` statements in several places, but you unfortunately miss one of them, you will receive an erroneous output. When multiple developers work on the same code base, this type of problem can occur easily. This is why though `lock` statements can make your code thread-safe, they may not always provide an ideal solution.

**Q3.10 In that case, how can you make a better solution?**

Notice that instead of working on a thread-unsafe data structure and guarding its usage by surrounding the `lock` statements by yourself, it would be better if you could directly work on the thread-safe data structures where the designers themselves ensure the thread-safety. The good news is that Microsoft provides you with this facility by providing you with concurrent collections, which is the topic of our discussion now.

# ConcurrentStack<T>

This section demonstrates how to use the `ConcurrentStack<T>` class in a program. It is a thread-safe last in-first out (LIFO) collection. You already know that its nonconcurrent equivalent is `Stack<T>`. Let's exercise some programming now.

The following code creates an instance of this class:

```
var stack = new ConcurrentStack<int>();
```

The `Push` method inserts an element at the top of the stack. Here is a sample where I push three elements as follows:

```
stack.Push(1);
stack.Push(2);
stack.Push(3);
```

The stack contains three elements now. Let's examine two other useful methods.

- The `TryPeek` method attempts to return the top element from the stack without removing it.
- However, the `TryPop` method tries to remove the top element from the stack. Otherwise, it will return `false`.

Let's test these methods by appending the following code segment:

```
int top;
if (stack.TryPeek(out top))
    WriteLine($"{top} is on top now.");

if (stack.TryPop(out top))
    WriteLine($"{top} is popped now.");

if (stack.TryPeek(out top))
    WriteLine($"{top} is on top now.");
```

At this stage, if you run this program, you can see the following output:

```
3 is on top now.
3 is popped now.
2 is on top now.
```

The output confirms that the `TryPeek` did not remove the top element, whereas the `TryPop` method was able to pop the top element (which is 3 in our example).

There is a method for efficient popping, which is called `TryPopRange`. This method attempts to pop and return multiple elements from the top of `ConcurrentStack<T>` atomically. The simplest version of this method has the following signature:

```
public int TryPopRange(T[] items)
```

You can see that the return type of the `TryPopRange` is not a boolean, but it is an `int`. Keeping this in mind, you can write the following code that tries to pop five elements from the top of the stack and displays the result (in this code segment, `items` is a placeholder to hold the elements):

```
var items = new int[5];
if (stack.TryPopRange(items) > 0)
{
    foreach (int item in items)
    {
        Write($"{item}\t");
    }
}
```

You may be thinking that since our stack has two elements now, trying to pop more than two elements at this moment can cause an exception. **However, this assumption is not correct!** If you run this code segment, you won't receive any exception; instead, you'll see the following output:

```
2       1       0       0       0
```

You can see that 2 and 1 were popped successfully. What about the remaining zeros? You guessed it right! They are the default values of the array cells.

However, the `TryPopRange` method has another overload that has the following signature:

```
public int TryPopRange(T[] items, int startIndex, int count)
```

The parameters have their usual meaning (you can learn about them at https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1.trypoprange?view=net-8.0). Using this overloaded version, if you write the following code segment:

```
var items = new int[5];
if (stack.TryPopRange(items, 1, 3) > 0)
{
    foreach (int item in items)
    {
        Write($"{item}\t");
    }
}
```

You'll see the following output:

```
0       2       1       0       0
```

Notice that this time the top three elements from the stack are popped, and they are placed in the `items` array starting from index 1.

**POINT TO NOTE**

However, while using this overloaded version of `TryPopRange`, you need to be careful. If `startIndex + count` is greater than the length of items, you'll receive the `System.ArgumentException` exception.

If you need to add many elements to the stack, pushing one element at a time may not be a good idea. In this case, you can use an efficient method, called `PushRange`. About this method, Microsoft (states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1.pushrange?view=net-8.0):

> ***...PushRange*** *guarantees that all of the elements will be added atomically, meaning that no other threads will be able to inject elements between the elements being pushed. Items at lower indices in the items array will be pushed before items at higher indices.*

To illustrate, let me create an array of three integers and push them to our current stack as follows:

```
var newElements = new int[] { 10, 11, 12 };
stack.PushRange(newElements);
```

If you put a breakpoint and debug the code, you'll see the expected result (see Figure 3-3).

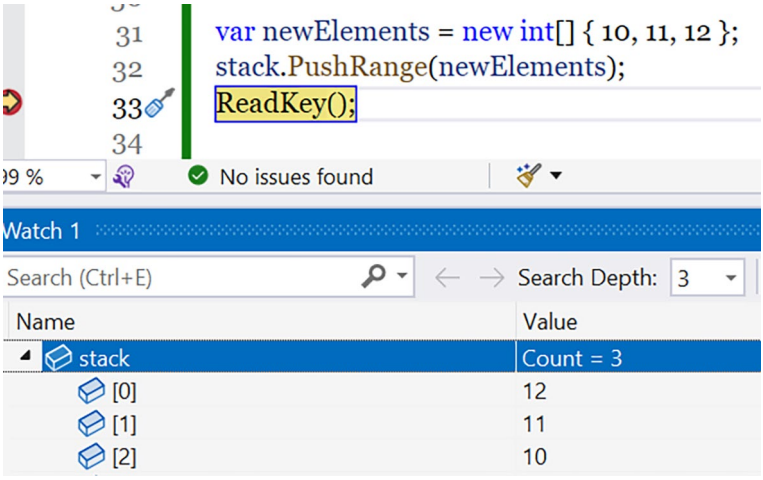



***Figure 3-3.*** *The integers 10, 11, and 12 are pushed to the stack*

---

**Note** You can download the project `UnderstandingConcurrentStack` to run the code segments that I discussed so far.

---

I assume you now have gotten an idea about the concurrent stack. **Since you are learning about parallel programming, let's now exercise a program where multiple tasks are involved in a multithreaded environment**. In this example, you'll see the following namespace:

```
using System.Collections.Concurrent;
```

This is because we are working on the types that are available in this namespace.

---

**Note** In fact, in all programs in this section, you will see this namespace.

---

Next, you will see me using the following code:

```
IProducerConsumerCollection<Car> cars = new
ConcurrentStack<Car>();
```

You can see that I used the polymorphic code while instantiating an instance of `ConcurrentStack<Car>`. Alternatively, I could instantiate this as follows: `ConcurrentStack<Car> cars = new ConcurrentStack<Car>();`. This approach could allow me to call class-specific methods such as `Push`, as well as `TryPop` on the `cars` instance. However, I did not opt for this approach because of the following reason:

- You can change the underlying data structure easily. For example, when I demonstrate to you the concurrent version of the `Queue` class, I will change only one line of code, which is as follows: `IProducer ConsumerCollection<Car> cars= new Concurrent Queue<Car>();`. There will be no other change in that demonstration.

Next, you'll see two different tasks (`addBlackCarModelsTask` and `addNonBlackCarModelsTask`) start running on separate threads. These tasks are used to push different `Car` instances where the `Car` class is defined as follows:

```
class Car(string model, string color)
{
    private string _model = model;
    private string _color = color;

    public override string ToString()
    {
        return $"[{_model},{_color}]";
    }
}
```

**POINT TO NOTE**

I have used a primary constructor in this class. This feature comes in C# 12. Here is the equivalent code that does not use the primary constructor in the `Car` class:

```
class Car
{
    private string _model;
    private string _color;
    public Car(string model, string color)
    {
        _model = model;
        _color = color;
    }
    public override string ToString()
    {
        return $"[{_model},{_color}]";
    }
}
```

You can learn more about primary constructors at https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/primary-constructors.

To make the program short and simple, I used three simple methods. They are as follows:

- The `AddBlackCarModels` method is used to push the cars that have the black color.
- The `AddNonBlackCarModels` method is used to push the cars that do not have the black color.

- The `RemoveCarModels` method is used to remove the cars from the collection.

Once the cars are added to the stack, a third task, called `removeCarsTask`, is created to exercise the `RemoveCarModels` method. You should not have any trouble understanding the rest of the program now.

## Demonstration 5

Here is the complete program:

**POINTS TO NOTE**

I want you to note the following points:

- To demonstrate the working mechanisms of different collections, I have used only a few cars in this demonstration and the upcoming demonstrations. So, you need not worry about the little bit of code duplication in these examples. In a real-life scenario, you can use common functionality to avoid code repetitions.
- For simplicity and to avoid more typing, I am using the terms *car models* and *cars* interchangeably. I hope that this will not trouble you much.

```
using static System.Console;
using System.Collections.Concurrent;

IProducerConsumerCollection<Car> cars = new
ConcurrentStack<Car>();

var addBlackCarModelsTask = Task.Run(AddBlackCarModels);
var addNonBlackCarModelsTask = Task.Run(AddNonBlackCarModels);
Task.WaitAll(addBlackCarModelsTask, addNonBlackCarModelsTask);
```

```
WriteLine($"A total of {cars.Count} cars are added to the
  repository.");

WriteLine("Removing the items now...");
var removeCarsTask = Task.Run(RemoveCarModels);
removeCarsTask.Wait();
WriteLine($"A total of {cars.Count} cars are present in the
  repository now.");
void AddNonBlackCarModels()
{
    Car car;
    car = new("Hyundai Creta", "Pearl");
    WriteLine($"Adding: {car}");
    cars.TryAdd(car);
    Thread.Sleep(1000);

    car = new("Maruti Suzuki Alto 800", "Red");
    WriteLine($"Adding: {car}");
    cars.TryAdd(car);
    Thread.Sleep(1000);

    car = new("Toyota Fortuner Avant", "Bronze");
    WriteLine($"Adding: {car}");
    cars.TryAdd(car);
    Thread.Sleep(1000);
}

void AddBlackCarModels()
{
    Car car;
    car = new("Toyota Fortuner Attitude", "Black");
```

```
    WriteLine($"Adding: {car}");
    cars.TryAdd(car);
    Thread.Sleep(1000);

    car = new("Hyundai Creta Abyss", "Black");
    WriteLine($"Adding: {car}");
    cars.TryAdd(car);
    Thread.Sleep(1000);

}

void RemoveCarModels()
{
    foreach (Car car in cars)
    {
        cars.TryTake(out Car result);
        WriteLine($"Tried removing: {result}");
    }

}

// Using primary constructor
class Car(string model, string color)
{
    private string _model = model;
    private string _color = color;
    public override string ToString()
    {
        return $"[{_model}, {_color}]";
    }
}
```

## Output

Here is some sample output:

```
Adding: [Toyota Fortuner Attitude, Black]
Adding: [Hyundai Creta, Pearl]
Adding: [Hyundai Creta Abyss, Black]
Adding: [Maruti Suzuki Alto 800, Red]
Adding: [Toyota Fortuner Avant, Bronze]
A total of 5 cars are added to the repository.
Removing the items now...
Tried removing: [Toyota Fortuner Avant, Bronze]
Tried removing: [Maruti Suzuki Alto 800, Red]
Tried removing: [Hyundai Creta Abyss, Black]
Tried removing: [Hyundai Creta, Pearl]
Tried removing: [Toyota Fortuner Attitude, Black]
A total of 0 cars are present in the repository now.
```

## Analysis

You can see that all five car models were added to the stack, but they were removed in a last in first out (LIFO) fashion.

# ConcurrentQueue<T>

The `ConcurrentQueue<T>` class is a thread-safe first in-first out (FIFO) collection. You already know that its nonconcurrent equivalent is `Queue<T>`. Earlier I told you if you are familiar with `Queue<T>`, you can easily understand `ConcurrentQueue<T>`. Let's verify some simple code segments that use some useful methods of this class.

The following code creates an instance of this class:

```
var queue = new ConcurrentQueue<int>();
```

While discussing the ConcurrentStack<T> class, you used the Push and TryPop methods. This time you will see the Enqueue and TryDequeue methods. The Enqueue method adds an element at the end of the queue (often called the *rear* end). Here is a sample of adding three elements:

```
queue.Enqueue(1);
queue.Enqueue(2);
queue.Enqueue(3);
```

The queue contains three elements now. Let's examine two other useful methods.

- The TryPeek method attempts to return the element from the beginning of the queue (often called the *front* end) without removing it.
- However, the TryDequeue method tries to remove the element from the front end. Otherwise, it will return false.

Let's test these methods by appending the following code segment:

```
int front;
if (queue.TryPeek(out front))
    WriteLine($"{front} is at the front now.");

if (queue.TryDequeue(out front))
    WriteLine($"{front} is removed now.");

if (queue.TryPeek(out front))
    WriteLine($"{front} is at the front now.");
```

At this stage, if you run this program, you can see the following output:

```
1 is at the front now.
1 is removed now.
2 is at the front now.
```

The output confirms that TryPeek did not remove the front element, whereas the TryDequeue method was able to remove the front element (which is 3 in our example).

---

**Note** You can download the project UnderstandingConcurrentQueue to run these code segments that I discussed so far.

---

Let's now exercise a program where multiple tasks are involved in a multithreaded environment.

## Demonstration 6

In the previous demonstration, you saw the following line:

```
IProducerConsumerCollection<Car> cars = new ConcurrentStack<Car>();
```

Replace this line with the following one (the key change is shown in bold):

```
IProducerConsumerCollection<Car> cars = new ConcurrentQueue<Car>();
```

There is no other change in this program. Since it is a small change, I do not show the entire program. You can download the project Demo6_ConcurrentQueueDemo to see the complete program.

## Output

Here is some sample output after running this program:

```
Adding: [Hyundai Creta, Pearl]
Adding: [Toyota Fortuner Attitude, Black]
Adding: [Maruti Suzuki Alto 800, Red]
Adding: [Hyundai Creta Abyss, Black]
Adding: [Toyota Fortuner Avant, Bronze]
```

```
A total of 5 cars are added to the repository.
Removing the items now...
Tried removing: [Hyundai Creta, Pearl]
Tried removing: [Toyota Fortuner Attitude, Black]
Tried removing: [Maruti Suzuki Alto 800, Red]
Tried removing: [Hyundai Creta Abyss, Black]
Tried removing: [Toyota Fortuner Avant, Bronze]
A total of 0 cars are present in the repository now.
```

### Analysis

You can see all five different car models were added to the queue, and later, they were removed in first in first out (FIFO) fashion.

## ConcurrentBag<T>

You have seen the concurrent versions of stacks and queues. However, if you are familiar with the concept of stack and queue data structure, you might have been a little bit bored. This time, let's learn about concurrent bags that do not guarantee the orderings. They are neither FIFO nor LIFO. Microsoft states the following about them (see https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1?view=net-8.0):

> *Bags are useful for storing objects when ordering doesn't matter, and unlike sets, bags support duplicates. ConcurrentBag<T> is a thread-safe bag implementation, optimized for scenarios where the same thread will be both producing and consuming data stored in the bag. ConcurrentBag<T> accepts null as a valid value for reference types.*

In the case of a concurrent stack, you saw the `Push`, `TryPop`, and `TryPeek` methods. In the case of a concurrent queue, you saw the `Enqueue`, `TryDequeue`, and `TryPeek` methods. In the case of a concurrent bag, you will see the `Add`, `TryTake`, and `TryPeek` methods in similar contexts.

However, before I show you the demonstration, let's understand the basic working mechanism. Let me tell you that it's a complex algorithm. (If you disagree with me, I suggest you see the implementation of the built-in method `TrySteal`.) However, I am trying to simplify this algorithm for you using some simple diagrams that you see shortly.

Before you see these diagrams, remember that internally each thread gets its own isolated storage (more specifically, its own private linked list). As a result, while adding the data, there is no conflict. In Figure 3-4, two threads add some items (using the `Add` method) in their own thread areas.

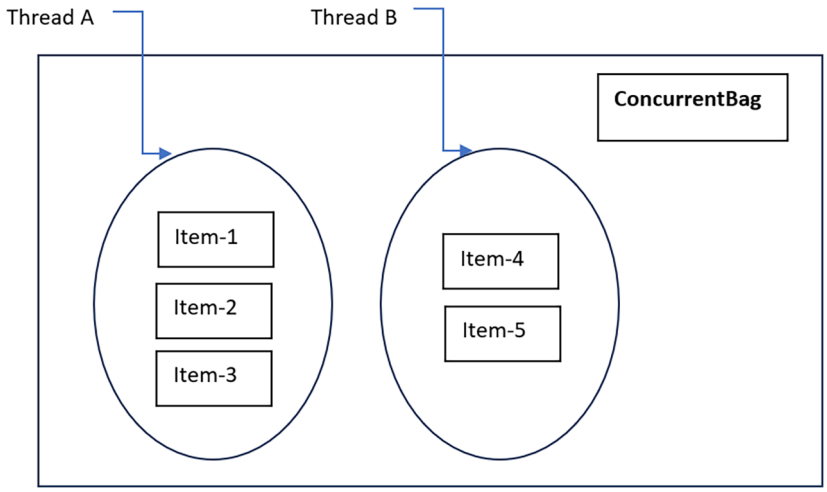



***Figure 3-4.*** *The concurrent bag's status after the addition of five items by Thread A and Thread B*

By calling the TryTake method, a thread can remove an element from its storage area. Let's assume, Thread A calls TryTake once and removes Item-3. Let's also assume that Thread B calls the TryTake method twice. As a result, both Item 4 and Item 5 will be removed (one-by-one) from their storage areas. Figure 3-5 depicts this situation after these removals.

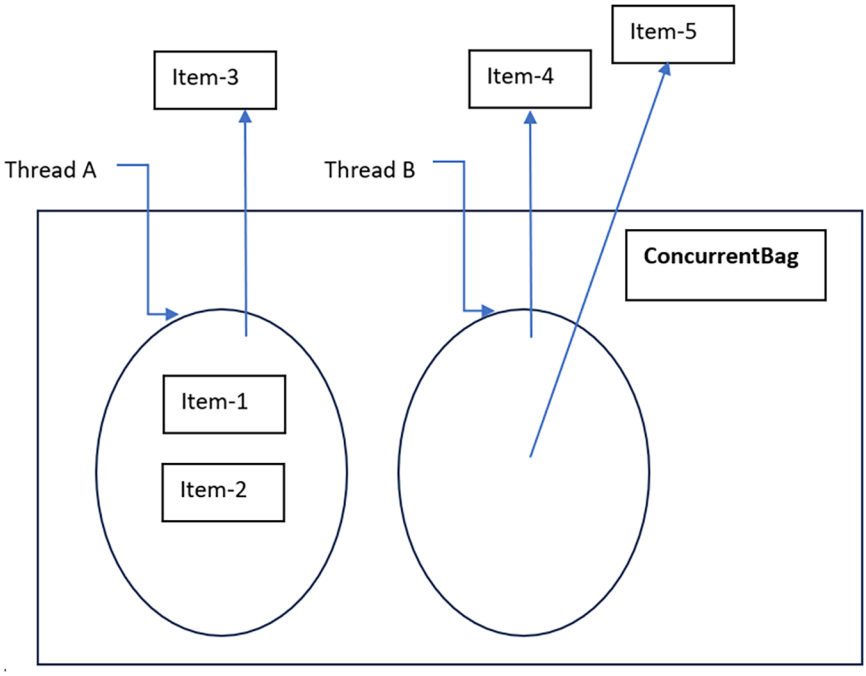



***Figure 3-5.*** *The concurrent bag's status after the removal of Item-3, Item-4, and Item-5*

Let's assume that Thread B calls the TryTake method once more. Since there is no item in its thread area, now it will try taking another item from its neighbor, i.e., Thread A's storage area (see Figure 3-6).

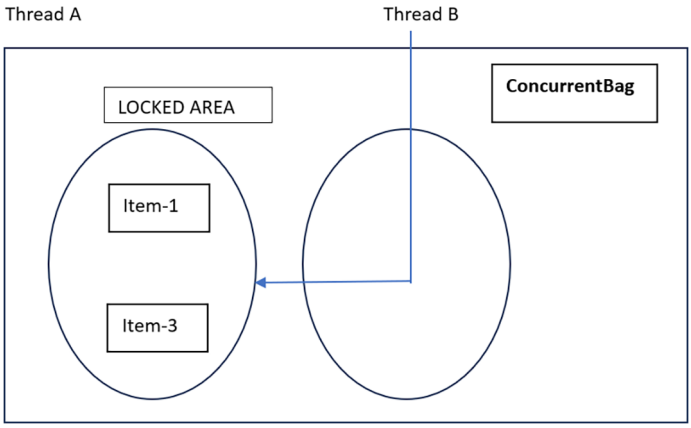



***Figure 3-6.*** *Thread B is stealing an item from Thread A*

While taking this item, Thread A's storage area will be locked (if you see the implementation of `TrySteal`, you can see the `Interlocked` operation). As a result, this time the execution will be slower. In this algorithm, since one thread steals an item from another thread, this is often called a **work-stealing algorithm**.

In the upcoming demonstration, you'll see me using the `TryTake` method. This method attempts to remove and return an object from the concurrent bag. If you investigate the implementation in Visual Studio, you'll see the following:

```
public bool TryTake([MaybeNullWhen(false)] out T result)
{
    WorkStealingQueue? queue =
    GetCurrentThreadWorkStealingQueue(forceCreate: false);
    return (
      queue != null &&
      queue.TryLocalPop(out result)) ||
      TrySteal(out result, take: true);
}
```

On further investigation, you'll learn that `TryLocalPop` removes an item from the tail of the queue, and `TrySteal` is a local helper method to steal an item from any other nonempty thread. I hope that things are making sense to you!

However, most of the time, a thread adds items to its own storage area and removes items from its own storage area. Since there is no need to employ locks in these cases, the executions are very fast.

I assume that you have got the idea! Let's see the demonstration now. In the previous demonstration, you saw the following line:

```
IProducerConsumerCollection<Car> cars = new ConcurrentQueue<Car>();
```

This time you'll see the following (the key change is shown in bold):

```
IProducerConsumerCollection<Car> cars = new ConcurrentBag<Car>();
```

Note that `ConcurrentBag<T>` implements the `TryAdd` method of the `IProducerConsumerCollection<T>` interface as follows:

```
bool IProducerConsumerCollection<T>.TryAdd(T item)
 {
     Add(item);
     return true;
 }
```

You can see that the `TryAdd` method internally calls the `Add` method. I used the `TryAdd` method in this demonstration to make it similar to the previous programs.

However, I slightly modified the methods. Earlier, you saw `AddBlackCarModels` and `AddNonBlackCarModels`. In that case, these `AddXXX` methods were performing the addition operations only; this time I renamed them to `ProcessXXX` (i.e., `ProcessBlackCarModels` and `ProcessNonBlackCarModels`). This is because to describe the working

mechanism of a concurrent bag, I want the respective tasks to perform both the addition and removal operations. In addition, I also display the task IDs to verify which task performed the addition or the removal.

## Demonstration 7

Here is the complete program. (There is no change in the Car class. It is not shown to avoid the repetition.)

```
using static System.Console;
using System.Collections.Concurrent;

IProducerConsumerCollection<Car> cars = new ConcurrentBag<Car>();

var addBlackCars = Task.Run(ProcessBlackCarModels);
var addNonBlackCars = Task.Run(ProcessNonBlackCarModels);
Task.WaitAll(addBlackCars, addNonBlackCars);
WriteLine($"At present, the repository contains {cars.Count}
 car(s).");

void ProcessNonBlackCarModels()
{
    Car car;
    car = new("Hyundai Creta", "Pearl");
    WriteLine($"Adding: {car} using task-{Task.CurrentId}");
    cars.TryAdd(car);
    Thread.Sleep(1000);

    car = new("Maruti Suzuki Alto 800", "Red");
    WriteLine($"Adding: {car} using task-{Task.CurrentId}");
    cars.TryAdd(car);
    Thread.Sleep(1000);
```

```
    car = new("Toyota Fortuner Avant", "Bronze");
    WriteLine($"Adding: {car} using task-{Task.CurrentId}");
    cars.TryAdd(car);
    Thread.Sleep(1000);

    WriteLine($"Task-{Task.CurrentId} will try removing one
     item now.");
    if (cars.Count > 0)
    {
        cars.TryTake(out Car removeCar);
        WriteLine($"Tried removing: {removeCar} using task-
         {Task.CurrentId}");
    }
}

void ProcessBlackCarModels()
{
    Car car;
    car = new("Toyota Fortuner Attitude", "Black");
    WriteLine($"Adding: {car} using task-{Task.CurrentId}");
    cars.TryAdd(car);
    Thread.Sleep(1000);

    car = new("Hyundai Creta Abyss", "Black");
    WriteLine($"Adding: {car} using task-{Task.CurrentId}");
    cars.TryAdd(car);

    // Putting a relatively long sleep so that the other
    // task can finish in between.
    Thread.Sleep(5000);

    WriteLine($"Task-{Task.CurrentId} will try removing three
     items now.");
```

```
    for (int i = 0; i < 3; i++)
    {
        if (cars.Count > 0)
         {
            cars.TryTake(out Car removeCar);
            WriteLine($"Tried removing: {removeCar} using task-
             {Task.CurrentId}");
        }
    }
}

// There is no change in the Car class. It is not shown to
// avoid the repetition.
```

## Output

Here is some sample output after running this program:

```
Adding: [Toyota Fortuner Attitude, Black] using task-1
Adding: [Hyundai Creta, Pearl] using task-2
Adding: [Hyundai Creta Abyss, Black] using task-1
Adding: [Maruti Suzuki Alto 800, Red] using task-2
Adding: [Toyota Fortuner Avant, Bronze] using task-2
Task-2 will try removing one item now.
Tried removing: [Toyota Fortuner Avant, Bronze] using task-2
Task-1 will try removing three items now.
Tried removing: [Hyundai Creta Abyss, Black] using task-1
Tried removing: [Toyota Fortuner Attitude, Black] using task-1
Tried removing: [Hyundai Creta, Pearl] using task-1
At present, the repository contains 1 car(s).
```

## Analysis

Notice that `task-1` added the black cars, and `task-2` added the nonblack cars. Later, `task-2` removed one car, and `task-1` removed three more cars. Here is the interesting part: while removing these cars, these tasks first attempt to remove the cars that were added by them. For example, `task-2` removed the car [Toyota Fortuner Avant, Bronze] because it was available in its private storage. However, `task-1` added only two black cars. This is why while removing, it first removed those black cars and then took another car from its neighbor and removed it as well.

Did you notice another interesting point? Each thread first tried to remove the items that were most recently added by them to their local list . However, while removing an item from a neighbor list, `task-1` removed the least recently added item from the neighbor's list.

**Author's Note:** Regarding the removal process, if interested, you can see the online discussion at https://stackoverflow.com/questions/78382286/trytake-is-stealing-an-element-that-was-added-least-recently-on-another-thread/78387452#78387452.

## Q&A Session

**Q3.11 How does `List<T>` differ from `ConcurrentBag<T>`?**

`List<T>` is ordered, whereas `ConcurrentBag<T>` is unordered. However, the important thing to understand is that `ConcurrentBag<T>` is thread-safe, but the other one is not thread-safe. In a concurrent environment, instead of manually implementing the locking mechanism with `List<T>`, it is often advantageous to use a built-in thread-safe construct like `ConcurrentBag<T>`.

## BlockingCollection<T>

Microsoft states the following about the `BlockingCollection<T>` class (see https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1?view=net-8.0):

> *Provides blocking and bounding capabilities for thread-safe collections that implement IProducerConsumerCollection<T>.*

Why do we need the blocking and bounding capabilities? Let me remind you that the core idea behind the producer-consumer pattern is that the producers produce some items and the consumers consume those items. However, the production and consumption rates can vary. As a result, you need to be prepared for the following situations:

- You cannot add an item if the underlying collection is full.
- You cannot remove an item if the underlying collection is empty.

This is why we need the blocking and bounding capabilities in a producer-consumer pattern.

Let's recollect how to use the built-in `TryAdd` and `TryTake` methods of the `IProducerConsumerCollection<T>` interface. When the `TryXXX` methods are unsuccessful, they return `false`. (For example, invoking the `TryTake` method on `ConcurrentQueue<T>` returns `false` if the queue is empty.)

However, sometimes you'd like the consumer to wait until an item is available or the producer to wait until a slot is available to place the produced item. How can you achieve this? You'll probably be tempted to make an overloaded version of these methods. However, the C# designers make your life easy! They have designed the `BlockingCollection<T>` type to support these scenarios.

Now the question is: how does `BlockingCollection<T>` help? It wraps any collection that implements `IProducerConsumerCollection<T>`. So, depending upon a user's choice, it can wrap any of the following types:

- `ConcurrentQueue<T>`
- `ConcurrentStack<T>`
- `ConcurrentBag<T>`

By default, it uses `ConcurrentQueue<T>`. As a result, now a consumer can take an item from the wrapped collection if an item is available. Otherwise, the consumer is blocked. Similarly, by limiting the size of the collection, you can block a producer when the collection is full. A collection limited in this manner is known as a **bounded blocking collection**.

**POINT TO NOTE**

The blocking collection is not itself a collection. It is just a wrapper around an underlying collection that implements the `IProducerConsumerCollection<T>` interface.

Let's see how to use the `BlockingCollection<T>` class. At the time of this writing, this class has the following constructors:

```
public BlockingCollection();
public BlockingCollection(IProducerConsumerCollection<T>
  collection);
public BlockingCollection(int boundedCapacity);
public BlockingCollection(IProducerConsumerCollection<T>
  collection, int boundedCapacity);
```

You can see that you can wrap a collection that implements `IProducerConsumerCollection<T>`. At the same time, you can also limit the size (see the `boundedCapacity` parameters).

While using the `BlockingCollection<T>` class, you need to remember the following:

- First, you instantiate the class. You can also specify the underlying collection along with the size. In the upcoming demonstration, I used the following line: `BlockingCollection<Car> cars = new(3);`, so, in this case, the `ConcurrentQueue<Car>` with size 3 will be instantiated. If you want a concurrent bag, you can write something like the following: `BlockingCollection<Car> cars = new(new ConcurrentBag<Car>(), 3);`.
- To add (i.e., produce) an element to the underlying collection, you can use the `Add` or `TryAdd` method. I have used the `Add` method in the upcoming example. In addition, I invoked the `CompleteAdding` method to put an end marker after which no element can be added to the collection. The consumers can monitor the `IsCompleted` property to know when this collection is empty or if no more additions will be accepted.
- Similarly, to remove (i.e., consume) an element, you can use the `Take` or `TryTake` method. However, when I consume the items, I have used `GetConsumingEnumerable`, which helps me to consume the elements when they are available. In other words, you block the task if there is no element to consume, but you resume the task when an element is available.

**BLOCKINGCOLLECTION<T>.ADD VS. BLOCKINGCOLLECTION<T>.TRYADD**

The documentation at https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1.add?view=net-8.0 states the following:

*If a bounded capacity was specified when this instance of BlockingCollection<T> was initialized, a call to Add may block until space is available to store the provided item.*

The documentation at https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1.tryadd?view=net-8.0 states the following:

*Tries to add the specified item to the BlockingCollection<T> within the specified time period, while observing a cancellation token.*

You can see that the `Add` method and the `TryAdd` method are similar except for the second one, you can use a timeout parameter. A similar difference can be observed between the `Take` and `TryTake` methods.

Now you should not face any difficulty in understanding the following program that produces 10 similar cars and consumes them in parallel. You may note that in this program, the production rate is much higher than the consumption rate (a production takes a minimum of 200 milliseconds, whereas a consumption takes a minimum of 500 milliseconds).

**POINT TO NOTE**

There are a few more interesting properties and methods available inside the `BlockingCollection<T>` class. You can use them to manage different scenarios such as cancellations and timeouts as well. In the following program, I did not use them. However, you can learn about those remaining constructs at https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1.-ctor?view=net-8.0.

## Demonstration 8

Here is the complete program (since there is no change in the `Car` class, it is not shown to avoid the repetition):

```
using static System.Console;
using System.Collections.Concurrent;

BlockingCollection<Car> cars = new(3);

var produceAndConsumeTask=Task.Run(ExerciseProducerConsumer);
produceAndConsumeTask.Wait();
WriteLine($"At present, the repository contains {cars.Count}
 car(s).");

void ExerciseProducerConsumer()
{
    var producerTask = Task.Run(ProduceCars);
    // Activating the producer a little bit early
    Thread.Sleep(300);
    var consumerTask = Task.Run(ConsumeCars);
    try
```

```
    {
        Task.WaitAll(producerTask, consumerTask);
    }
    catch (Exception ex)
    {
        WriteLine(ex.ToString());
    }
}

void ProduceCars()
{
    Car car;
    for(int i = 0;i<10;i++)
    {
        car = new($"{i}-Hyundai Creta", "Pearl");
        cars.Add(car);
        Thread.Sleep(200);
        WriteLine($"Produced: {car}");

    }
   // The following code prevents the consumer's foreach loop
   // from hanging
    cars.CompleteAdding();
    WriteLine($"+++ Production completed. +++");
}
void ConsumeCars()
{
    foreach (var car in cars.GetConsumingEnumerable())
    {
        WriteLine($"\tConsumed: {car}");
        Thread.Sleep(500);
    }
```

```
    WriteLine($"--- Consumption completed. ---");
}

// There is no change in the Car class. It is not shown to
// avoid the repetition.
```

## Output

Here is some sample output after running this program:

```
Produced: [0-Hyundai Creta, Pearl]
        Consumed: [0-Hyundai Creta, Pearl]
Produced: [1-Hyundai Creta, Pearl]
Produced: [2-Hyundai Creta, Pearl]
        Consumed: [1-Hyundai Creta, Pearl]
Produced: [3-Hyundai Creta, Pearl]
Produced: [4-Hyundai Creta, Pearl]
        Consumed: [2-Hyundai Creta, Pearl]
Produced: [5-Hyundai Creta, Pearl]
        Consumed: [3-Hyundai Creta, Pearl]
Produced: [6-Hyundai Creta, Pearl]
        Consumed: [4-Hyundai Creta, Pearl]
Produced: [7-Hyundai Creta, Pearl]
        Consumed: [5-Hyundai Creta, Pearl]
Produced: [8-Hyundai Creta, Pearl]
        Consumed: [6-Hyundai Creta, Pearl]
Produced: [9-Hyundai Creta, Pearl]
+++ Production completed. +++
        Consumed: [7-Hyundai Creta, Pearl]
        Consumed: [8-Hyundai Creta, Pearl]
        Consumed: [9-Hyundai Creta, Pearl]
--- Consumption completed. ---
At present, the repository contains 0 car(s).
```

## Analysis

Notice that though the production rate is much higher than the consumption rate, you never see more than three cars produced before the consumption starts. The **bounded blocking collection** blocked the producer task when three elements were already available to consume.

**Author's Note:** If the production rate is very fast, in some rare cases, it may appear to you that more than three cars are produced before the consumption starts. However, the consumption already started before you see that the "extra" car is produced. Ideally, you should never see a higher number of cars (such as five, six, or more) available before the consumption starts.

# Note from Microsoft

Since `BlockingCollection<T>` implements the `IDisposable` interface, Microsoft suggests you dispose of the type once you finish the work. To dispose of this directly, you can call the `Dispose` method inside a `try/catch` block. You can also dispose of the type indirectly via the language construct `using`. However, you should remember that the `Dispose` method is not thread-safe. To make the examples short and simple, I have ignored the concept of garbage collection in these programs.

# ConcurrentDictionary<TKey,TValue>

This section demonstrates how to use the `ConcurrentDictionary<TKey,TValue>` class in a program. It is a thread-safe collection of key/value pairs on which multiple threads can work concurrently. You already know that its nonconcurrent equivalent is `Dictionary<TKey,TValue> Stack<T>`.

Let's create a dictionary that contains countries with the most FIFA World Cup titles. The following code creates an instance of this class:

```
var fifaWCTitles = new ConcurrentDictionary<string, int>();
```

You will see methods for adding and removing elements. But before that, I'd like to mention that you can assign values for the key indexes as follows:

```
fifaWCTitles["Brazil"] = 5;
fifaWCTitles["Germany"] = 4;
fifaWCTitles["Argentina"] = 2; // Will be updated to 3 later
```

If you debug the code, you can see that the dictionary contains these three elements. Figure 3-7 shows an example.

***Figure 3-7.*** *The dictionary contains three elements.*

To add an element to the dictionary, you can use the `TryAdd` method as well. This method attempts to add an element to the dictionary with the specified key and value. If the key already exists, this method returns `false`. At present, our dictionary does not have the key `"Italy"`, so, the following code can add one more item to the dictionary:

```
fifaWCTitles.TryAdd("Italy", 4);
```

However, since the record of Argentina is already added, if you write something like the following:

```
fifaWCTitles.TryAdd("Argentina", 3); // will not update
```

it will not update the value for Argentina.

Then how do you update the value? You can try the `AddOrUpdate` method. The following code uses this method where you pass the existing key and its value for the first two parameters and a lambda expression as the third parameter:

```
fifaWCTitles.AddOrUpdate("Argentina", 2, (key, oldValue) => 3);
```

Using the code intelligence in Visual Studio, you can see that this lambda expression is used to generate a new value for an existing key based on the key's existing value.

To illustrate, if you replace the previous line of code with the following code segment:

```
fifaWCTitles.AddOrUpdate(
    "Argentina",
     2,
    (key, oldValue) =>
        {
            int newValue = 3;
            WriteLine($"{key}'s old value {oldValue} is
              updated to new value {newValue}.");
            return newValue;
        }
 );
```

executing this code segment, you can see the following output:

```
Argentina's old value 2 is updated to new value 3.
```

Let's check the current dictionary status in the debug mode (see Figure 3-8).

***Figure 3-8.*** *The dictionary contains four elements with the updated record for Argentina*

You can see that the dictionary now contains four elements with the updated value for Argentina.

`GetOrAdd` is another useful method. If the key does not exist, using this method, you can add a new element to the dictionary as follows:

```
fifaWCTitles.GetOrAdd("France", 2); // Successful
```

If the key is present, this method returns the existing value for the key; otherwise, it will return the new value of the key. So, if I need to check the return value, I can write something like this:

```
int value=fifaWCTitles.GetOrAdd("France", 2); // Successful
WriteLine(value);
```

Anyway, you can easily understand that after adding the details of France, if you now write the following:

```
fifaWCTitles.GetOrAdd("France", 3); // Unsuccessful
```

it will not update the record.

OK, you have seen how to add elements to a concurrent dictionary or update an element in this dictionary. Let's try the removal operation as well. You guessed it right! Like the `TryAdd` method, there is a `TryRemove` method as well. The `TryRemove` method attempts to remove and return the value with the specified key from the dictionary. In the following code segment, I tried removing the records for France and Uruguay:

```
TryRemovingCountryDetails("France");
TryRemovingCountryDetails("Uruguay");

void TryRemovingCountryDetails(string country)
{
 var isRemoved = fifaWCTitles.TryRemove(country, out int
  removedValue);
 if (isRemoved)
 {
  WriteLine($"{country} with {removedValue} World Cup titles is
   deleted.");
 }
 else
 {
  WriteLine($"Could not remove the details of {country}");
 }
}
```

Since the key "France" is present in our dictionary, its record can be deleted. However, the dictionary did not contain any information about the key "Uruguay". So, you'll see the following output:

```
France with 2 World Cup titles is deleted.
Could not remove the details of Uruguay
```

---

**Note** Download the UnderstandingConcurrentDictionary project to run these code segments that I discussed so far.

---

## Are Atomicity and Thread-Safety Guaranteed?

For the update (and write) operations, the concurrent dictionary uses fine-grained locking to ensure thread safety. However, the documentation says the following about the built-in methods (see https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2?view=net-8.0):

> *All these operations are atomic and are thread-safe with regards to all other operations on the ConcurrentDictionary<TKey,TValue> class. The only exceptions are the methods that accept a delegate, that is, AddOrUpdate and GetOrAdd. For modifications and write operations to the dictionary,*

If you further scroll down the page, you will see the following warning message as well:

> *All public and protected members of ConcurrentDictionary<TKey,TValue> are thread-safe and may be used concurrently from multiple threads. However, members accessed through one of the interfaces the ConcurrentDictionary<TKey,TValue> implements, including extension methods, are not guaranteed to be thread safe and may need to be synchronized by the caller.*

Remember this information while using a concurrent dictionary in your program.

I assume you have some idea about the concurrent dictionary now. Since you are learning parallel programming, let's now exercise a program where multiple tasks are involved in a multithreaded environment.

## Demonstration 9

The following is the complete program. (There is no change in the `Car` class. It is not shown here to avoid the repetition.)

```
using static System.Console;
using System.Collections.Concurrent;
```

```
ConcurrentDictionary<int, string> cars = new();

var addBlackCarModelsTask = Task.Run(AddBlackCarModels);
var addNonBlackCarModelsTask = Task.Run(AddNonBlackCarModels);
Task.WaitAll(addBlackCarModelsTask, addNonBlackCarModelsTask);
WriteLine($"A total of {cars.Count} cars are added to the
 repository.");

WriteLine("\nTrying to remove one item at random now.");
int random = new Random().Next(1, 6);
var removeCarsTask = Task.Run(()=>RemoveOneCar(random));
removeCarsTask.Wait();
WriteLine($"\nA total of {cars.Count} cars are present in the
 repository now.");
WriteLine("\nHere is the current content of the dictionary:");
foreach (var car in cars)
{
    WriteLine($"The key: {car.Key} has the value: {car.Value}");
}

void AddNonBlackCarModels()
{
    Car car;
    car = new("Hyundai Creta","Pearl");
    WriteLine($"Adding: {car}");
    cars.TryAdd(1,car.ToString());
    Thread.Sleep(1000);

    car = new("Maruti Suzuki Alto 800", "Red");
    WriteLine($"Adding: {car}");
    cars.TryAdd(2, car.ToString());
    Thread.Sleep(1000);
```

```
    car = new("Toyota Fortuner Avant", "Bronze");
    WriteLine($"Adding: {car}");
    cars.TryAdd(3, car.ToString());
    Thread.Sleep(1000);
}

void AddBlackCarModels()
{
    Car car;
    car = new("Toyota Fortuner Attitude", "Black");
    WriteLine($"Adding: {car}");
    cars.TryAdd(4, car.ToString());
    Thread.Sleep(1000);

    car = new("Hyundai Creta Abyss", "Black");
    WriteLine($"Adding: {car}");
    cars.TryAdd(5, car.ToString());
    Thread.Sleep(1000);
}

void RemoveOneCar(int key)
{
    var isRemoved = cars.TryRemove(key, out string removedValue);
    if (isRemoved)
    {
        WriteLine($"The key {key} with value: {removedValue} is
         deleted.");
    }
    else
    {
        WriteLine($"Could not remove the key: {key}");
    }
}
```

```
// There is no change in the Car class. It is not shown to
// avoid the repetition.
```

## Output

Here is some sample output:

```
Adding: [Hyundai Creta, Pearl]
Adding: [Toyota Fortuner Attitude, Black]
Adding: [Hyundai Creta Abyss, Black]
Adding: [Maruti Suzuki Alto 800, Red]
Adding: [Toyota Fortuner Avant, Bronze]
A total of 5 cars are added to the repository.

Trying to remove one item at random now.
The key 4 with value: [Toyota Fortuner Attitude, Black] is
deleted.

A total of 4 cars are present in the repository now.

Here is the current content of the dictionary:
The key: 1 has the value: [Hyundai Creta, Pearl]
The key: 2 has the value: [Maruti Suzuki Alto 800, Red]
The key: 3 has the value: [Toyota Fortuner Avant, Bronze]
The key: 5 has the value: [Hyundai Creta Abyss, Black]
```

## Analysis

You can see that at the beginning, two different tasks added five different cars simultaneously to the dictionary. Later, one car (that had key 4) was deleted at random.

## Q&A Session

**Q3.12 Should I always prefer concurrent collections over their counterpart?**

It depends on the requirements of the application. As a general guideline, Microsoft suggests the following (see https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/):

> *When you write new code, use the concurrent collection classes to write multiple threads to the collection concurrently. If you're only reading from a shared collection, then you can use the classes in the* `System.Collections.Generic` *namespace.*

In the end, to make the code efficient, you want to use methods that can serve you best. You may remember that concurrent collections have some special methods (the `TryPop` method) that can help you perform atomic test-and-act operations that can provide you benefits in a multithreaded environment.

# Exercises

**E3.1** Can you predict the output of the following code?

```
using static System.Console;
using System.Collections.Concurrent;

var colorBag = new ConcurrentDictionary<int,string>();
colorBag.TryAdd(1, "Red");
colorBag.TryAdd(2, "Green");
string color=colorBag.GetOrAdd(2, "Blue");
WriteLine(color);
```

**E3.2** Can you predict the output of the following code?

```
using static System.Console;
```

```
var resetEvent = new AutoResetEvent(true);

WriteLine("Two customers are approaching the mall.");
Task.Run(VisitMall);
Task.Run(VisitMall);

// Imposing some delay to mimic a real-world scenario
Thread.Sleep(2000);

WriteLine("Press any key to exit.");
ReadKey();

void VisitMall()
{
    // Imposing a small delay to mimic a real-world scenario
    Thread.Sleep(1000);
    WriteLine($"The customer {Task.CurrentId} is waiting for
     the entry pass.");
    resetEvent.WaitOne();
    WriteLine($"The customer {Task.CurrentId} enters the mall.");
}
```

**E3.3** Let's make some changes to the previous program (E3.2) as follows (the changes are shown in comments):

```
using static System.Console;

// For Exercise 3.2
// var resetEvent = new AutoResetEvent(true);

// For Exercise 3.3
var resetEvent = new AutoResetEvent(false);
// Invoking the set method twice in advance
resetEvent.Set();
resetEvent.Set();

// There is no other change in the remaining code
```

You can see that the main thread invoked the Set method twice in advance. Will it help to release two upcoming worker threads immediately?

**E3.4** Can you predict the output of the following code?

```
using static System.Console;
using System.Collections.Concurrent;

var stack = new ConcurrentStack<int>();
for(int i=1;i<=5;i++)
{
    stack.Push(i);
}
var items = new int[10];
if (stack.TryPopRange(items,1, 10) > 0)
{
    foreach (int item in items)
    {
        Write($"{item}\t");
    }
}
```

**E3.5** Suppose an organization wants to fill its vacant posts. So, it announces an advertisement saying that interested people can apply for these posts by filling up an online form. However, each candidate needs to make an online payment of $5 to the organization's account number. You understand that many people can fill up the online form and pay the mentioned amount simultaneously. Assume that at a particular moment, 1,000 people tried to make the online payment. As a result, the organization should get $5000 in its account online. Can you write a program to mimic the scenario and verify the result?

**E3.6** Demonstration 4 showed you an example of one-way signaling where the worker thread waited for a signal from the main thread. Now I want you to write a program that deals with two-way signaling using `AutoResetEvent`. You can consider a known scenario: during the billing process, any customer can raise a complaint. As a result, the billing process is paused until the issue is resolved. Can you write a program that mimics this scenario?

**E3.7** Demonstration 8 in this chapter showed how to use a blocking collection. Can you update this program that allows you to pass a cancellation request as well?

# Summary

This chapter started with a discussion on synchronization. It divided the synchronization constructs into four categories and showed you examples from each category including the `lock` statements, the `Interlocked` class, and signaling using `AutoResetEvent`. It also covered some blocking methods (this also belongs to the synchronization categories) such as `Thread.Sleep` and `Task.Wait` in various program segments.

Then the chapter focused on the concurrent collections and covered the important types such as `ConcurrentStack<T>`, `ConcurrentQueue<T>`, `ConcurrentBag<T>`, `BlockingCollection<T>`, and `ConcurrentDictionary<TKey,TValue>`. It also covered the `IProducerConsumerCollection<T>` interface and discussed the possible implementations of the producer-consumer pattern. You may find the chapter a little bit lengthy, but I sincerely believe that these discussions will help you a lot.

# Solutions to Exercises

Here is a sample solution set for the exercises in this chapter.

## E3.1

You will see the following output:

```
Green
```

**Clue**: Since key 2 was already added in the dictionary, you are seeing the existing value as `Green`, but not as `Blue`.

## E3.2

This code will allow only one customer to enter the mall. Here is the sample output:

```
Two customers are approaching the mall.
The customer 2 is waiting for the entry pass.
The customer 1 is waiting for the entry pass.
The customer 2 enters the mall.
Press any key to exit.
```

(**Clue**: One customer was released immediately because `AutoResetEvent` was already in the signaled state.)

## E3.3

No. Here is some sample output:

```
Two customers are approaching the mall.
The customer 2 is waiting for the entry pass.
The customer 1 is waiting for the entry pass.
The customer 2 enters the mall.
Press any key to exit.
```

(**Clue**: It does not matter how many times you release a signal in advance; when the threads appear, only one of them will be released immediately. This issue was discussed in the Q&A sessions; see the answer to Q3.7.)

## E3.4

You'll see the following exception:

```
Unhandled exception. System.ArgumentException: The sum of the
startIndex and count arguments must be less than or equal to
the collection's Count.

// Other details such as line numbers etc. are not shown
```

## E3.5

Here is a sample program:

```
using static System.Console;

Account account = new();
var tasks = new List<Task>();
for(int i=0;i<1000;i++)
{
    tasks.Add
     (
        Task.Run( () => account.Credit(5))
     );
}
Task.WaitAll([.. tasks]);
// Same as:
// Task.WaitAll(tasks.ToArray());
```

```
WriteLine($"The current balance is ${account.Balance}");

// The organization's account
class Account
{
    private int _balance;
    public int Balance
    {
        get { return _balance; }
        set { _balance = value; }
    }
    public void Credit(int amount)
    {
        Interlocked.Add(ref _balance, amount);
    }
    // The following method is not used in this program
    public void Debit(int amount)
    {
        Interlocked.Add(ref _balance, -amount);
    }

}
```

This program produces the following output:

```
The current balance is $5000
```

## E3.6

Here is the sample program:

```
using static System.Console;

var billingEvent = new AutoResetEvent(false);
var complaintEventIfAny = new AutoResetEvent(false);
```

```
var billingTask = Task.Run(PrepareBill);

// Let the billing counter open first.
Thread.Sleep(2000);
WriteLine("The customer picks up the items and moves towards
  the billing counter.");

// Imposing some delay to mimic a real-world scenario
Thread.Sleep(1000);
billingEvent.Set();
WriteLine("The management waits if there is any complaint.");
complaintEventIfAny.WaitOne();
WriteLine("The management makes the customer happy.");

billingTask.Wait();
// Exception handling is not considered in this program

void PrepareBill()
{
    WriteLine("**The billing counter is waiting to serve the
     customer(s).**");
    billingEvent.WaitOne();
    WriteLine("**The billing counter starts functioning...**");
    int random = new Random().Next(1, 10);
    if (random > 3)
    {
        WriteLine($"The customer complains about a bill amount of
         ${random}");
    }
   complaintEventIfAny.Set();
    // Imposing a small delay to mimic a real-world scenario
    Thread.Sleep(2000);
    WriteLine("**The bill is generated.**");
}
```

Here is the sample output when the customer raises a complaint:

```
**The billing counter is waiting to serve the customer(s).**
The customer picks up the items and moves towards the billing
  counter.
**The billing counter starts functioning...**
The management waits if there is any complaint.
The customer complains about a bill amount of $9
The management makes the customer happy.
**The bill is generated.**
```

Here is more sample output when the customer does not raise a complaint:

```
**The billing counter is waiting to serve the customer(s).**
The customer picks up the items and moves towards the billing
  counter.
The management waits if there is any complaint.
**The billing counter starts functioning...**
The management makes the customer happy.
**The bill is generated.**
```

## E3.7

I want you to do this exercise yourself.

CHAPTER 4

# Working on Parallel Loops

C# developers are quite familiar with using `foreach` and `for` loops. By using these iteration statements, we can repeatedly execute a statement or a block of statements. This chapter primarily focuses on their parallel versions. It also helps you experiment with similar constructs that support parallel programming.

## Revisiting Sequential Loops

Before we discuss parallel loops, let's analyze a common scenario where you would usually use a loop. For example, given the code `List<int> numbers = Enumerable.Range(1, 10).ToList();`, can you increase each number by 100 and display the result? Yes, you can do this in several ways. For example, you can use any of the following three approaches.

In the first approach, you use a `foreach` loop as follows:

```
// Approach: 1
foreach (int i in numbers)
{
  WriteLine($"Number:{i}, Incremented number: {i + 100}");
}
```

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5_4

If you are a fan of the `for` loop, you can write something as follows (though this is less readable than the previous):

```
// Approach: 2
for (int i = 0; i < numbers.Count; i++)
{
    WriteLine($"Number:{numbers[i]}, Incremented number:
     {numbers[i] + 100}");
}
```

If you love the functional coding style, you can write the following equivalent code as well:

```
// Approach: 3
Numbers
 .ForEach
  (
    i => WriteLine($"Number:{i}, Incremented number: {i + 100}")
  );
```

I have shown how to use a `foreach` loop, a `for` loop, and a `Foreach` function in these approaches. However, notice that each approach follows the sequential execution.

# Experimenting with Parallel Loops

Processing data sequentially does not matter much if the data set is small. However, if you need to loop through a large amount of data, sticking to the sequential approach is not a wise decision. The reason is obvious: it is time-consuming. So, to speed it up, you may want to achieve parallelism by segregating this data set into several parts and executing those parts in parallel. However, the good news is that you have built-in constructs that support the parallel versions of `For` and `ForEach` as well.

# Introducing the Parallel Class

The `Parallel` class is made to provide these supports. Let's investigate this class. If you type `Parallel` and add a dot, you will see the screen in Figure 4-1.

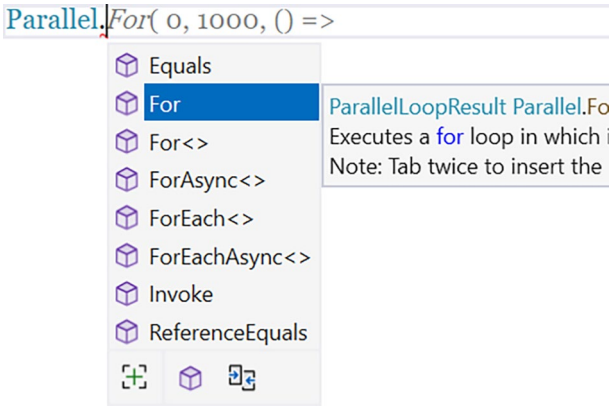



***Figure 4-1.*** *The methods of the Parallel class*

Figure 4-1 shows you the presence of the following methods: `For`, `ForEach`, `ForAsync`, `ForEachAsync`, and `Invoke` in the `Parallel` class. These are the methods we'll focus on in this chapter. Since I have not discussed the `async` and `await` keywords yet, in this chapter, we'll cover the following:

- **Parallel.ForEach** (this is the parallel equivalent of the C# `foreach` loop)
- **Parallel.For** (this is the parallel equivalent of the C# `for` loop)
- **Parallel.Invoke** (this executes an array of delegates in parallel)

**POINT TO REMEMBER**

In the context of parallel programming, a few terms are often seen in different books, blogs, and/or articles. If you are unaware of those terms, understanding the topic or participating in a discussion forum can be challenging. It is obvious that once you finish this book, you'll have a fair idea about many of them. For now, let me remind (or introduce) you to a few common terms: you have seen task parallelism constructs in Chapter 1 and Chapter 2. Now you'll see data parallelism. The Task Parallel Library (TPL) supports data parallelism through the `System.Threading.Tasks.Parallel` class. The `Parallel` class with those task parallelism constructs is known as Task Parallel Library (TPL). In Chapter 3, you saw the concurrent collections, and in Chapter 5, you'll learn Parallel LINQ (PLINQ). In Appendix A, I'll talk about PFX, which can be written as TPL + PLINQ + concurrent collections.

## Parallel.ForEach

Let's start with the parallel `ForEach`. There are many overloads of the `Foreach` method in the `Parallel` class (at the time of this writing, there are 20 overloads). Consider the following version that takes only two parameters:

```
public static ParallelLoopResult ForEach<TSource>(
 IEnumerable<TSource> source,
 Action<TSource> body)
{
    // Some code
}
```

As a result, given the code `List<int> numbers = Enumerable.Range(1, 10).ToList();`, you can now display the numbers using the parallel version of the `foreach` loop as follows:

```
// Approach: 4
Parallel.ForEach(
   numbers,
   i => WriteLine($"Number: {i}")
);
```

## Parallel.For

At the time of this writing, there are 12 overloads for the parallel `for` loop. Take a look at the following form:

```
public static ParallelLoopResult For(int fromInclusive,
 int toExclusive, Action<int> body)
{
 // Some code
}
```

The first argument indicates the starting point (inclusive), and the next argument is used for the endpoint (exclusive). For example, to display the numbers between 1 to 10, you can write the following code:

```
// Approach: 5
Parallel
 .For
(
  1,
  11,
  i => WriteLine($"Number: {i}")
 );
```

## Demonstration 1

Let's see a simple demonstration. While revisiting the sequential approaches, I incremented the numbers from 1 to 10 by 100. Now you have seen the parallel versions of the `foreach` and `for` loops. Let's verify whether these parallel loops work for us. In this program, I also print the task IDs to verify whether different tasks participate in this computation. Here is the complete program:

---

**Note** In this demonstration, I have shown you the parallel loops only. However, by downloading the `Demo1_WorkingOnParallelLoops` project, you can see their sequential versions as well. I have kept all of them in the same place for your ready reference.

---

```
using static System.Console;

List<int> numbers = Enumerable.Range(1, 10).ToList();

#region Parallel loops
WriteLine("\nUsing Parallel ForEach");
// Approach: 4
Parallel.ForEach(
   numbers,
   i => WriteLine($"Number:{i}, Incremented number:{i + 100},
        Id:{Task.CurrentId}")
);
WriteLine("Ending Parallel ForEach");

// Approach: 5
WriteLine("\nUsing Parallel For");
Parallel.For(
```

```
        1,
        numbers.Count + 1,
        i => WriteLine($"Number:{i}, Incremented number:{i + 100},
             Id:{Task.CurrentId}")
    );
WriteLine("Ending Parallel For");
#endregion
```

## Output

Here is some sample output:

```
Using Parallel ForEach
Number:7, Incremented number:107, Id:4
Number:5, Incremented number:105, Id:3
Number:1, Incremented number:101, Id:1
Number:2, Incremented number:102, Id:1
Number:4, Incremented number:104, Id:1
Number:8, Incremented number:108, Id:4
Number:6, Incremented number:106, Id:3
Number:3, Incremented number:103, Id:2
Number:9, Incremented number:109, Id:5
Number:10, Incremented number:110, Id:1
Ending Parallel ForEach

Using Parallel For
Number:3, Incremented number:103, Id:8
Number:9, Incremented number:109, Id:11
Number:10, Incremented number:110, Id:11
Number:2, Incremented number:102, Id:11
Number:6, Incremented number:106, Id:11
Number:4, Incremented number:104, Id:8
Number:7, Incremented number:107, Id:10
```

```
Number:1, Incremented number:101, Id:7
Number:5, Incremented number:105, Id:9
Number:8, Incremented number:108, Id:11
Ending Parallel For
```

## Analysis

The output confirmed the following characteristics:

- Different tasks participated in this program.
- Invoking the `ForEach` or `ForAll` method of the `Parallel` class caused these tasks to start automatically.
- If you keep executing this program, you'll always notice that the application finishes only after printing these numbers. This gives you a clue that these methods block until the work is complete.

# Parallel.Invoke

The `Parallel` class has another useful method, called `Invoke`. The documentation states the following (see https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff963549(v=pandp.10)?redirectedfrom=MSDN):

> ***Parallel.Invoke*** *is the simplest expression of the parallel task pattern. It creates new parallel tasks for each delegate method that is in its* ***params*** *array argument list. The* ***Invoke*** *method returns when all the tasks are finished.*

At the time of this writing, there are two overloads for the parallel `Invoke` method. Let me consider the following:

```
public static void Invoke(params Action[] actions)
```

Seeing this definition, you understand that we can pass a variable number of `Action` instances to the `Invoke` method. Let me create three such instances and pass them in the following program.

## Demonstration 2

Here is the complete demonstration:

```
using static System.Console;

#region Parallel.Invoke
WriteLine("Using Parallel.Invoke method.");
Action greet = new(() => WriteLine($"Task {Task.CurrentId}
 says: Hello reader!"));
Action printMsg = new(() => WriteLine($"Task {Task.CurrentId}
 says: This is a beautiful day."));
Action suggest = new(() => WriteLine($"Task {Task.CurrentId}
 says: Relax now."));
Parallel.Invoke(greet,printMsg,suggest);
WriteLine("End Parallel.Invoke");
#endregion
```

## Output

Here is some sample output:

```
Using Parallel.Invoke method.
Task 3 says: Hello reader!
Task 1 says: This is a beautiful day.
Task 2 says: Relax now.
End Parallel.Invoke
```

## Q&A Session

**Q4.1 I can see that task 1 finishes after task 3. Is this the expected behavior? I can also see that you did not wait for the tasks to be finished. Is this OK?**

Yes. The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.invoke?view=net-8.0):

> *This method can be used to execute a set of operations, potentially in parallel. No guarantees are made about the order in which the operations execute or whether they execute in parallel. This method does not return until each of the provided operations has completed, regardless of whether completion occurs due to normal or exceptional termination.*

**Q4.2 It appears to me that I could create separate tasks and wait for them to finish their executions instead of using `Parallel.Invoke`. Is this correct?**

Nice observation. The documentation confirms the following (see https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff963549(v=pandp.10)?redirectedfrom=MSDN):

> *Internally, **Parallel.Invoke** creates new tasks and waits for them. It uses methods of the **Task** class to do this.*

If you study the mentioned link, you will understand that I could write an equivalent program as follows:

```
using static System.Console;

Task  greet2 = Task.Factory.StartNew(
() => WriteLine($"Task {Task.CurrentId} says: Hello reader!"));

Task printMsg2 = Task.Factory.StartNew(
() => WriteLine($"Task {Task.CurrentId} says: This is a
beautiful day."));
```

```
Task suggest2 = Task.Factory.StartNew(
() => WriteLine($"Task {Task.CurrentId} says: Relax now."));

Task.WaitAll(greet2, printMsg2, suggest2);
```

However, notice that when you work with a large number of delegates, creating separate tasks for each delegate and managing them is not a good idea. Using `Parallel.Invoke` can help you! It will work efficiently in those cases as well.

**Q4.3 When we use a sequential `for` loop, the step size need not be 1. For example, given the numbers 0 to 50, I can print the numbers 0, 5, 10, 15, ..., 45, and 50 using the following code:**

```
int step = 5;
for (int i = 0; i <= 50; i += step)
{
    Write($"{i}\t");
}
```

**How can I write an equivalent code using a parallel loop?**

You can define a custom method. Here is a sample code fragment that uses a custom method, called `InputDomain`, that can be used to traverse the elements with a preferred step size (in this example, it is 5).

```
static IEnumerable<int> InputDomain(int start, int
endInclusive, int stepCounter)
{
    for (int i = start; i <= endInclusive; i += stepCounter)
    {
        yield return i;
    }
}
```

Having this function, I can now use `Parallel.Foreach` to print the numbers as follows:

```
Parallel.ForEach(
    InputDomain(0, 50, 5),
    i => Write($"{i}\t")
   );
```

You can download the project `Q&A_ParallelForWithCustomStep` to execute the complete program. Here is some sample output that was generated by executing this code:

```
15      5       0
35      40      45      50      10      20      25      30
```

Finally, you can also use the following code to get an equivalent output:

```
Parallel.ForEach(
 Enumerable
   .Range(0, 11)
   .Select(i => i * 5),
    i => Write($"{i}\t")
   );
```

# Scenarios for Parallel Execution

Each scenario is not suitable for parallel execution. For example, if you execute a parallel `for` loop for a small number of iterations or use it to process a small data set, you may not see the benefit. In those cases, you may also notice that parallel execution is slower. To illustrate this, let's analyze the following case studies.

## Case Study 1

Consider the numbers from 1 to 10. Now calculate the square of these numbers and display the result. Let's investigate the time taken when we use the following:

- A sequential `foreach` loop and
- A parallel `foreach` loop

## Demonstration 3

Here is a sample program:

```
using System.Diagnostics;
using static System.Console;

WriteLine("Examining sequential versus parallel execution.");
List<int> input= Enumerable.Range(1, 10).ToList();

ExecuteSequentialFor(input);
ExecuteParallelFor(input);
void ExecuteSequentialFor(List<int> numbers)
{
    Stopwatch sw = Stopwatch.StartNew();

   foreach (int i in numbers)
    {
        Write($"{i * i}\t");
    }
    sw.Stop();
    WriteLine($"\nSequential execution(using foreach) time:
     {sw.ElapsedMilliseconds} ms\n");
}
```

```
WriteLine("_______________");
void ExecuteParallelFor(List<int> numbers)
{
    Stopwatch sw = Stopwatch.StartNew();
    Parallel.ForEach(
          numbers,
          i => Write($"{i * i}\t")
        );
    sw.Stop();
    WriteLine($"\nParallel execution(using Parallel.ForEach)
     time: {sw.ElapsedMilliseconds} ms");
}
```

## Output

I executed this program several times. In each case, my computer showed that sequential execution was faster. Here is some sample output:

```
Examining sequential versus parallel execution.
1     4     9     16    25    36    49    64    81    100
Sequential execution(using foreach) time: 0 ms

81    49    25    36    4     16    64    9     1     100
Parallel execution(using Parallel.ForEach) time:20 ms
```

# Case Study 2

Now assume that in addition to printing the numbers, we are doing some other work as well. To mimic this activity, let me impose a small delay inside the loops and replace the previous loops (that are shown in Demonstration 3) with the following loops:

```
// New sequential loop

 foreach (int i in numbers)
  {
      // Imposing some delay to do some other work
      Thread.Sleep(500);
      Write($"{i * i}\t");
  }

// New Paralllel loop

Parallel.ForEach(
       numbers,
       i =>
       {
           // Imposing some delay to do some other work
           Thread.Sleep(500);
           Write($"{i * i}\t");
       }
     );

// There is no other change in the program
```

Run this modified program now. Here is some sample output:

```
Examining sequential versus parallel execution.
1      4      9      16     25     36     49     64     81     100
Sequential execution(using forech) time: 5093 ms

1      25     81     9      49     16     64     100    36     4
Parallel execution(using Parallel.ForEach) time:1081 ms
```

Since each iteration sleeps for 500 ms, there is no surprise that the sequential execution will take more than 5000 ms. **However, notice that this time parallel execution is almost five times faster than the sequential execution.**

## Analysis

There is no doubt that the output can vary in your system. Still, from these case studies, the following points are evident:

- Not every scenario is suitable for parallel execution.
- A sequential execution can be faster than a parallel execution if the data set is small.
- On the other hand, when you work on a large data set or perform independent time-consuming (and blocking) operations, preferring parallel execution over sequential execution can provide you benefits.

**POINT TO NOTE**

Surely, parallelization, like any multithreaded code, will inherit complexities. It is because a multithreaded environment can be affected by many other factors such as thread creation, ensuring thread safety, workload distribution, context switching time, and so on. However, you'll learn that you can further fine-tune the parallel executions. I'll discuss this soon.

# Useful Parameters

As mentioned, there are different overloads of `Parallel.For` and `Parallel.ForEach`. We have looked at some case studies with parallel `foreach` loops already. So, this time we'll do an upcoming analysis with parallel `for` loops.

Figure 4-2 shows the different parameters of `Parallel.For`.

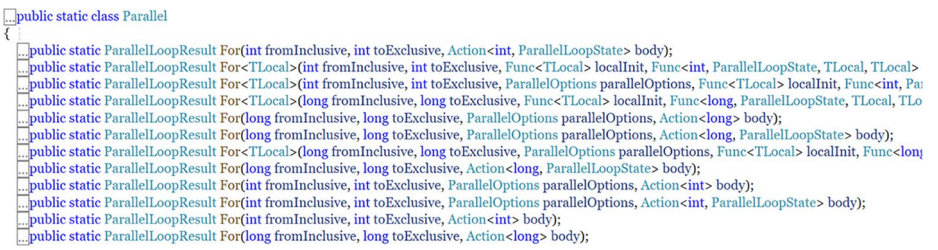



***Figure 4-2.*** *Different overloads of Parallel.For*

Figure 4-2 shows the presence of many different parameters. Most of them are very common, but there are a few new entries as well. Let's focus on the following two parameters:

- `ParallelLoopState`
- `ParallelOptions`

Understanding these parameters can help you use the parallel loops effectively.

# Using ParallelLoopState

Using the `ParallelLoopState` parameter, you can monitor and/or manipulate the loop. I'll show you the usage of the following methods:

- `Stop`
- `Break`

## Using Stop

Here is a sample demonstration where I tried to stop the loop at the seventh iteration using the `Stop` method. (You'll see that the `Break` method is commented inside this program. After we analyze the output, I'll comment out the `Stop` method and uncomment the `Break` method.)

## Demonstration 4

Here is a sample demonstration:

```
using static System.Console;

var repeatedTask = Task.Run(() =>
{
    Parallel.For(
       1,
       16,
        (int i, ParallelLoopState state) =>
        {
            if (i == 7)
            {
                WriteLine($"Processing: {i} ID:{Task.
                 CurrentId}. About to interrupt the loop.");
                state.Stop();
               // state.Break();
            }
            Thread.Sleep(1000);
            WriteLine($"Processed: {i} ID:{Task.CurrentId}");
        });
});

repeatedTask.Wait();
WriteLine("End");
```

## Output

The output can vary on different machines. However, let me show you some sample output:

```
Processing: 7 ID:2. About to interrupt the loop.
Processed: 1 ID:4
Processed: 4 ID:3
Processed: 10 ID:5
Processed: 7 ID:2
End
```

## Using Break

Replace `Stop()` with `Break()` in the previous program now. Here is some sample output after this change (the output may vary on your computer):

```
Processing: 7 ID:4. About to interrupt the loop.
Processed: 1 ID:2
Processed: 4 ID:3
Processed: 10 ID:5
Processed: 7 ID:4
Processed: 2 ID:7
Processed: 5 ID:8
Processed: 3 ID:2
Processed: 6 ID:8
End
```

## Q&A Session

**Q4.4 How `Break` is different from `Stop`?**

The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallelloopstate.break?view=net-8.0):

> *Break indicates that no iterations after the current iteration should be run. It effectively cancels any additional iterations of the loop. However, it does not stop any iterations that have already begun execution. For example, if Break is called from*

> *the 100th iteration of a parallel loop iterating from 0 to 1,000, all iterations less than 100 should still be run, but the iterations from 101 through to 1000 that have not yet started are not executed.*

From the given output, you can verify this as well. You can see that when I called the `Break` method from iteration number 7, the numbers from 1 to 6 were processed successfully. But when I used the `Stop` method, this was not the case. In the output you can see the presence of 1 and 4, but `{1,4}` is only a subset of `{1,2,3,4,5,6}`. When you use the `Stop` method, this is an expected behavior.

Now you understand that when `state.Break()` is called, the loop ceases execution at the system's earliest convenience of all iterations beyond the current iteration. It does not stop the current or prior iterations. It's more like "Don't schedule any more iterations, but let the ones that have already started finish."

However, after invoking `state.Stop()`, the loop ceases execution of any more iterations at the system's earliest convenience, even if they are prior to the current iteration. It's more like "Stop all iterations as soon as possible."

In summary, `Break` is a polite request to wind down the loop after the current iteration, while `Stop` is a more immediate request to halt the loop.

## Retrieving More Information

The parallel versions of `for` and `foreach` methods return a `ParallelLoopResult` object that exposes two useful properties called `IsCompleted` and `LowestBreakIteration`. They help you identify whether the loop ran to completion; if it did not, it can point to the cycle where it was broken. Here is a short description of them:

- `IsCompleted`: This property returns true if all iterations of the loop were executed, and false if the loop was exited early, by a call to either `Break` or `Stop`.

- `LowestBreakIteration`: This property returns the index of the first iteration of the loop for which `Break` was called. If `Break` was not called, this property returns `null`. This property is useful for understanding how much of the loop was completed before it was broken.

Let's see an example. While using the `Break` method, if you use the following version (changes are in bold) of the `repeatedTask` in Demonstration 4:

```
var repeatedTask = Task.Run(() =>
{
    ParallelLoopResult result =
    Parallel.For
    (
       1,
       16,
        (int i, ParallelLoopState state) =>
        {
            if (i == 7)
            {
                WriteLine($"Processing: {i} ID:{Task.
                 CurrentId}. About to interrupt the loop.");
                //state.Stop();
                state.Break();
            }
            Thread.Sleep(1000);
            WriteLine($"Processed: {i} ID:{Task.CurrentId}");
        }
     );
    WriteLine($"Loop completed? {result.IsCompleted}");
    // Retrieving more info if the loop is incomplete
    if (!result.IsCompleted)
```

```
    {
        if (result.LowestBreakIteration != null)
        {
            WriteLine($"The lowest break iteration:
             {result.LowestBreakIteration}");
        }
        else
        {
            WriteLine("The Stop() method was called.");
        }
    }
});
```

You can see an output like the following (notice the important changes in bold):

```
Processing: 7  ID:3. About to interrupt the loop.
Processed: 1  ID:2
Processed: 4  ID:4
Processed: 10  ID:5
Processed: 7  ID:3
Processed: 2  ID:7
Processed: 3  ID:2
Processed: 5  ID:8
Processed: 6  ID:8
Loop completed? False
The lowest break iteration: 7
End
```

But if you use the `Stop` method (instead of the `Break` method), you can see an output like the following (notice the important changes in bold):

```
Processing: 7 ID:4. About to interrupt the loop.
Processed: 4 ID:3
```

```
Processed: 1 ID:2
Processed: 10 ID:5
Processed: 7 ID:4
Loop completed? False
The Stop() method was called.
End
```

---

**Note** Download the `Demo4_ParallelLoopState_Modified` project to see the complete program from the Apress website.

---

## Using ParallelOptions

Let's discuss the `ParallelOptions` class. This class has three useful properties:

- `TaskScheduler`: This property gets or sets the `TaskScheduler` that is used to schedule the tasks. The `TaskScheduler` is responsible for managing how and when tasks are executed. You may also note that by default methods on the `Parallel` class are non-cancelable, use the default task scheduler (`TaskScheduler.Default`), and attempt to use all the available processors. By using `ParallelOptions`, you can override these defaults.
- `MaxDegreeOfParallelism`: This property gets or sets the maximum number of concurrent tasks enabled by the `ParallelOptions` instance. It allows you to limit the number of concurrent operations run by `Parallel` methods, which can be useful to prevent overloading the system with too many tasks.

- `CancellationToken`: This property gets or sets the `CancellationToken` associated with this `ParallelOptions` instance. A `CancellationToken` allows the cooperative cancellation of the parallel operation. It can be used to request that the operation be canceled, although it's up to the operation to check the cancellation token and respond appropriately.

Since .NET provides a default `TaskScheduler` instance, normally you do not need to worry about it. Let's focus on the other two parameters. I'll start the discussion with `MaxDegreeOfParallelism`.

By executing the following code on my computer:

```
using static System.Console;

var repeatedTask2 = Task.Run(() =>
{
    Parallel.For(
        1,
        10,
        i =>
        {
            WriteLine($"Processed:{i} with the task
             ID:[{Task.CurrentId}]");
        });
});

repeatedTask2.Wait();
```

I see the following sample output:

```
Processed:5 with the task ID:[4]
Processed:6 with the task ID:[4]
Processed:8 with the task ID:[4]
Processed:9 with the task ID:[4]
```

```
Processed:2 with the task ID:[4]
Processed:4 with the task ID:[4]
Processed:7 with the task ID:[5]
Processed:1 with the task ID:[2]
Processed:3 with the task ID:[3]
```

It is not a surprise to see different task IDs. The four task IDs represent four different tasks in this output. Interestingly, you can control the number of tasks. How? Let's see the following demonstration.

## Demonstration 5

In the following program, I create an instance of `ParallelOptions` where I limit the maximum number of concurrent tasks to 2 and pass it to the parallel `for` loop as follows:

```
using static System.Console;

ParallelOptions parallelOptions = new()
{
    MaxDegreeOfParallelism = 2
};

var repeatedTask2 = Task.Run(() =>
{
  Parallel.For(
   1,
   10,
   parallelOptions,
   i =>
   {
```

```
        WriteLine($"Processed:{i} with the task ID: [{Task.
         CurrentId}]");
    });
});

repeatedTask2.Wait();
```

## Output

Here is some sample output that I got by executing this program:

```
Processed:5 with the task ID:[2]
Processed:1 with the task ID:[3]
Processed:6 with the task ID:[2]
Processed:7 with the task ID:[2]
Processed:8 with the task ID:[2]
Processed:2 with the task ID:[3]
Processed:3 with the task ID:[3]
Processed:4 with the task ID:[3]
Processed:9 with the task ID:[2]
```

## Analysis

This time there are only two different task IDs present in this output. The official documentation says the following (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.paralleloptions.maxdegreeofparallelism?view=net-8.0#system-threading-tasks-paralleloptions-maxdegreeofparallelism):

> *A positive property value limits the number of concurrent operations to the set value. If it is -1, there is no limit on the number of concurrently running operations (with the exception of the ForEachAsync method, where -1 means ProcessorCount).*

Now you understand that you can set the value as -1, but not -2, -3, and so forth. It's a design decision.

You have seen the usefulness of the `Paralleloptions` class where I used `MaxDegreeOfParallelism.` The `MaxDegreeOfParallelism` property in .NET can be set to -1, indicating no limit on concurrent tasks, with the system optimizing based on available resources. Any other negative value is invalid and will cause a `System.ArgumentOutOfRangeException` to occur. You can use the `CancellationToken` property to manage cancellations. It is coming next.

# Managing Cancellations

In a concurrent environment, managing tasks is an important activity. Since any task can exercise a parallel loop, you may want to know how to cancel a parallel loop.

## Following the Previous Approach

Using the knowledge you learned in Chapter 2, you may attempt to cancel the task that executes a parallel loop as follows:

```
using static System.Console;

CancellationTokenSource cts = new();
CancellationToken token = cts.Token;

#region The previous approach that you learned in Chapter 2
var repeatedTask = Task.Run(() =>
{
    Parallel.For(
        1,
        50,
        (i) =>
```

```
        {
            token.ThrowIfCancellationRequested();
            WriteLine($"Processed: {i}");
            Thread.Sleep(1000);
        });
}, token);
#endregion
// Some other code (for example, you can initiate a
// cancellation or wait for the task to finish its execution)
```

## Recommended Approach

Though the previous code works, it is interesting to note that there are alternative approaches as well. To cancel a parallel loop, the official documentation recommends the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-cancel-a-parallel-for-or-foreach-loop?redirectedfrom=MSDN):

> *In a parallel loop, you supply the CancellationToken to the method in the ParallelOptions parameter and then enclose the parallel call in a try-catch block.*

Following Microsoft's suggestion, I create a `ParallelOptions` instance and pass this instance inside `repeatedTask` in the next demonstration. I also display the corresponding task IDs in this program.

## Demonstration 6

Here is the complete program with the important changes in bold:

```
using static System.Console;
```

```
CancellationTokenSource cts = new();
CancellationToken token = cts.Token;

#region The recommended approach

ParallelOptions parallelOptions = new()
{
    CancellationToken = token
};

var repeatedTask = Task.Run(() =>
{
    Parallel.For
    (
        1,
        50,
        parallelOptions,
        i =>
        {
            WriteLine($"Processed ={i}  ID:{Task.CurrentId}");
            Thread.Sleep(1000);
        }
    );

});
#endregion

try
{

    WriteLine("Press c to cancel");
    var input = ReadKey().KeyChar;
    if (input.Equals('c'))
        cts.Cancel();
```

```
    repeatedTask.Wait(token);
}

catch (OperationCanceledException oce)
{
    WriteLine($"\nCaught error due to cancellation: {oce.
     Message} ");
}

catch (AggregateException ae)
{
    ae.Handle(e =>
    {
        WriteLine($"\nCaught error:  {e.Message}");
        return true;
    });
}

finally
{
    cts.Dispose();
}

WriteLine("The application ends now.");
```

## Output

Here is some sample output that triggered a cancellation while the application was running:

```
Press c to cancel
Processed: 25  ID:5
Processed: 1  ID:3
Processed: 37  ID:4
```

```
Processed: 13  ID:2
Processed: 49  ID:6
Processed: 14  ID:2
Processed: 2  ID:9
Processed: 38  ID:8
Processed: 26  ID:10
c
Caught error due to cancellation: The operation was canceled.
The application ends now.
```

## Q&A Session

**Q4.5 In this demonstration, you have used `Wait(token)` instead of `Wait()`. Is there any specific thought behind this design?**

In this demonstration, the use of the `Wait(token)` method makes more sense to me. It also shows a similar pattern of outputs if you do not follow the recommended approach (for canceling the parallel loop).

However, if you replace this method with the `Wait` method and **do not follow the recommended approach**, you may see output like the following:

```
Press c to cancel
Processed: 1
Processed: 13
Processed: 37
Processed: 25
Processed: 49c
Caught error:  One or more errors occurred. (The operation was
canceled.) (The operation was canceled.) (The operation was
canceled.) (The operation was canceled.) (The operation was
canceled.)

The application ends now.
```

You can see lots of cancellations in this output. Does this look good? For me, the answer is no because only one message was sufficient to understand that something was canceled in between the executions.

---

**Note** I placed a sample project, named `Demo_RepeatingSameError`, in the `Chapter4` folder to give you an idea about this kind of output. You can download this project from the Apress website. You may also note that the output may vary based on your computer.

---

# Handling Exceptions

The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-handle-exceptions-in-parallel-loops):

> *The Parallel.For and Parallel.ForEach overloads do not have any special mechanism to handle exceptions that might be thrown. In this respect, they resemble regular for and foreach loops (For and For Each in Visual Basic); an unhandled exception causes the loop to terminate as soon as all currently running iterations finish.*

This is why while executing a parallel loop, you must handle the possible exceptions.

In Chapter 2, you learned about handling exceptions. You learned that you can wrap the possible exceptions in a `System.AggregateException`. The same idea applies here as well. I will now add a few lines of code to the previous demonstration to show you a modified program.

# Demonstration 7

Here I will apply some dummy logic to generate an exception. The logic is as follows:

- If the processing number is greater than 45, I generate a random number. If this random number is an even number, I generate an exception.

**POINT TO NOTE**

In this example scenario, I am throwing the exception based on a random value. This makes the discussion easy. I often use a similar logic in different programs to demonstrate various concepts. However, since the `Random` class is not thread-safe, in a typical application, you may need to use a `lock` statement surrounding the random value. However, this approach will limit the concurrency. So, a better idea is to generate separate `Random` instances for each thread using the `ThreadLocal<T>` class. You'll see a sample program that uses `Threadlocal<T>` shortly.

Here is the complete program:

```
using static System.Console;

CancellationTokenSource cts = new();
CancellationToken token = cts.Token;

ParallelOptions parallelOptions = new()
{
    CancellationToken = token
};
```

```
var repeatedTask = Task.Run(() =>
{
    Parallel.For(
        1,
        50,
        parallelOptions,
        i =>
        {
            WriteLine($"Processed: {i}  ID:{Task.CurrentId}");
            Thread.Sleep(1000);
            // A dummy logic to generate an exception randomly
            if (i > 45)
            {
                int random = new Random().Next(1, 11);
                if (random % 2 == 0)
                {
                    throw new Exception($" While processing the
                      number {i}, the random value was: {random}");
                }
            }
        });
});

try
{
    // Let us allow the loop to be executed for some time
    Thread.Sleep(1000);
    WriteLine("Press c to cancel");
    var input = ReadKey().KeyChar;
    if (input.Equals('c'))
        cts.Cancel();
```

```
     repeatedTask.Wait(token);
}

catch (AggregateException ae)
{
    ae.Handle(e =>
    {
        WriteLine($"\nCaught error:  {e.Message}");
        return true;
    });
}

catch (OperationCanceledException oce)
{
    WriteLine($"\nCaught error due to cancellation: {oce.
     Message} ");
}
finally
{
    cts.Dispose();
}

WriteLine("\nThe application ends now.");
```

## Output

Once again, you may see different output on your computer. However, I am including some sample output for your reference.

```
The application ends now.
Processed: 1  ID:14
Processed: 37  ID:16
Processed: 25  ID:15
Processed: 13  ID:13
```

```
Processed: 49  ID:17
Press c to cancel
Processed: 38  ID:20
Processed: 26  ID:19
Processed: 14  ID:13
Processed: 2  ID:21
Processed: 4  ID:18
Caught error: One or more errors occurred. ( While processing
the  number 49, the random value was: 8)

The application ends now.
```

You may note that I *did not* trigger a cancellation request while running the application. I pressed the Enter key only after the program stopped printing the messages in the console window.

# Fine-Tuning

You can fine-tune a parallel loop using some built-in constructs. Let's use the concept of a parallel `for` loop to calculate the factorial of 10. Once we analyze the output, you'll see its fine-tuned version.

**POINT TO NOTE**

You may wonder why we need to use a parallel loop to calculate this small computation. It is because opting for a parallel loop can be beneficial for certain scenarios such as when you are handling a large amount of data (say one million) or you are dealing with some time-consuming operations. However, consider the fact that since I am showing you the output of each demonstration, printing those large numbers of data is not only boring, it is time-consuming as well. So, I considered this simple example to show you a possible implementation, and later I'll fine-tune it.

To understand the code, read the following points:

- In this program, the variable `factorial` is shared among multiple threads/tasks.
- Since I exercise a parallel loop, I need to provide a synchronization mechanism to access this variable. To ensure this, I used a `lock` statement before updating this variable.

You should not have any trouble understanding the following code now:

```
using System.Diagnostics;
using static System.Console;

int n = 10;
object lockObject = new();
var stopWatch1 = Stopwatch.StartNew();
int factorial = 1;
Parallel.For(
    1,
    n + 1,
    (x) =>
    {
        lock (lockObject)
        {
            factorial *= x;
        }
    }
);
stopWatch1.Stop();
WriteLine($"The factorial of {n} is {factorial}");
```

```
WriteLine($"The elapsed time (Ticks): {stopWatch1.
ElapsedTicks}");
WriteLine("_____________");
```

Once I execute this program, my computer shows the following output:

```
The factorial of 10 is 3628800
The elapsed time (Ticks): 334679
```

## Using Thread-Local Variables

Let's fine-tune this program. Notice that in the previous demonstration, I needed to use a `lock` statement to provide the synchronization mechanism where only one thread can update the `factorial` variable at a particular time. However, this approach suffers a typical problem when each thread tries to access the shared variable many times.

How can you limit the access to this variable? You can calculate a partial product on each thread, and then you can work on those partial products inside the `lock` statement. This approach helps you avoid the overhead of synchronizing a large number of accesses to the shared variable. As a result, you will improve the overall performance of the application.

To implement this idea, I used the thread-local data in the following program. Let's see what Microsoft says about it. The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-parallel-for-loop-with-thread-local-variables):

> *By using thread-local data, you can avoid the overhead of synchronizing a large number of accesses to shared state. Instead of writing to a shared resource on each iteration, you compute and store the value until all iterations for the task are complete. You can then write the final result once to the shared resource, or pass it to another method.*

Now I use a different overload of the `Parallel.For` loop that can use the thread-local variable. It has the following signature:

```
public static ParallelLoopResult For<TLocal>(
    int fromInclusive,
    int toExclusive,
    Func<TLocal> localInit,
    Func<int, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally
)
```

Let's understand the parameters:

As usual, the **first two parameters** specify the starting and ending points for the iteration values.

The **third parameter** represents the function delegate that returns the initial state of the local data for each task. Notice that `localInit` is a `Func<TResult>` type where `TResult` is the type of the variable that will store the thread-local state. Its type is defined by the generic type argument supplied when you call the generic version of this `For` loop. You understand that in our example, it is `Int32`. The type argument tells the compiler the type of the temporary variable that will be used to store the thread-local state. In this example, the expression `() => 1` initializes the thread-local variable to one.

The **fourth parameter** defines the logic of this loop. Since its signature is (`Func<int, ParallelLoopState, TLocal, TLocal>`), I can use a lambda expression as follows:

```
(int num, ParallelLoopState state, int subFact) =>
{
    return num * subFact;
}
```

Here the first parameter (`num`) is the value of the loop counter for that iteration of the loop. The second parameter is a `ParallelLoopState` object (`state`) that can be used to break out of the loop. To make the example simple, I did not use the `state` variable in this example. The third parameter is the thread-local variable. Notice that the return type is also an `int` here.

The return value of this lambda iteration is used to initialize `subFact` on each subsequent iteration of the loop. You can consider this last parameter as a value that is passed to each iteration and then passed to the `localFinally` delegate when the last iteration is complete.

The **fifth parameter** defines the method that is called once after all the iterations on a particular thread have been completed. You'll see me using the `lock` statement here. As a result, I could avoid writing to the `factorial` variable multiple times on each iteration of the loop.

## Demonstration 8

Let's take a look at the complete program and measure the elapsed time for this fine-tuned program:

```
using System.Diagnostics;
using static System.Console;

int n = 10;
object lockObject = new();
WriteLine($"Calculating the factorial of a number.");

var stopWatch2 = Stopwatch.StartNew();
int factorial2 = 1;

Parallel.For(
    1,
    n + 1,
    () => 1,
```

```
    (int num, ParallelLoopState state, int subFact) =>
    {
        return num * subFact;
    },
   subFact =>
   {
       lock (lockObject)
       {
           factorial2 *= subFact;
       }
   }
 );
stopWatch2.Stop();
WriteLine($"Using thread-local data, the factorial of {n} is
 {factorial2}");
WriteLine($"After fine-tuning, the elapsed time is {stopWatch2.
 ElapsedTicks} timer ticks.");
```

## Output

Here is some sample output:

```
Using thread-local data, the factorial of 10 is 3628800
After fine-tuning, the elapsed time is 41516 timer ticks.
```

You can see that **this fine-tuned version is almost seven times faster than the previous program**. Though the output may vary, using thread-local data inside a parallel `for` loop can help you improve the performance of an application.

## Using Partition-Local Variables

When I was discussing `Parallel.For`, you saw the term **thread-local variable**. If you work with `Parallel.ForEach`, you may see the term **partition-local variable**. The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-parallel-foreach-loop-with-partition-local-variables):

> *When a ForEach loop executes, it divides its source collection into multiple partitions. Each partition has its own copy of the partition-local variable. A partition-local variable is similar to a thread-local variable, except that multiple partitions can run on a single thread.*

Using the concept that you learned in Demonstration 8, you should not find trouble while using partition-local variables inside the `Parallel.ForEach` method. I leave this exercise to you.

Note that I used this concept while solving exercise E4.6. If needed, you can refer to that program or follow the previously mentioned link to get more clarity on the topic.

## Additional Note

The scenario in Demonstration 8 can be handled by Parallel LINQ (PLINQ) as well. Once you learn about it in Chapter 5, you can write the following equivalent code:

```
int n = 10;
#region Using PLINQ

var stopWatch3 = Stopwatch.StartNew();
int factorial3 = ParallelEnumerable
    .Range(1, 10)
    .AsParallel()
```

```
    .Aggregate(1, (subFact, nxtNumber) => subFact * nxtNumber);
stopWatch3.Stop();
WriteLine($"Using PLINQ, the factorial of {n} is {factorial3}");
WriteLine($"Now the elapsed time is {stopWatch3.ElapsedTicks}
  timer ticks.");
#endregion
```

Here is some sample output that I got by running this code:

```
Using PLINQ, the factorial of 10 is 3628800
Now the elapsed time is 26709 timer ticks.
```

Notice that the code is both simple and efficient. In fact, it produced the best performance so far. However, it is obvious that the performance may vary on your computer.

PLINQ has a rich set of functionalities that can help you perform all the necessary steps that you'd like to do to parallelize your work. For example, PLINQ can partition the work into tasks, execute those tasks on different threads, and gather the results into a single output sequence. So, using PLINQ, you can avoid the burden of partitioning the work and output collating from different threads.

---

**Note** When you download the project `Demo8_UsingThreadLocalData`, you'll see that I have kept all the code segments: the first one does not use the thread-local data, the second one uses the thread-local data, and the final one uses PLINQ. As a result, you can compare the results instantly on your computer.

---

## Q&A Session

**Q4.6 It can help me if you explain thread-local storage with a simple example.**

If you think about Demonstration 7 and Demonstration 8, you'll understand that to limit the access to the shared variable, I needed to fine-tune the application. How did I do this? While making the computation on each thread, I wanted to keep the data isolated. In simple words, while one thread was doing the computation, I needed to prevent other threads from interfering with it. This is why I needed to consider thread-local storages/variables.

It is useful to note that you can implement thread-local storage in different ways. Some of the common approaches in this context include marking a static field with the `ThreadStaticAttribute` using `ThreadLocal<T>` class and using `AsyncLocal<T>` (useful for asynchronous programming).

I recommend you refer to https://learn.microsoft.com/en-us/dotnet/standard/threading/thread-local-storage-thread-relative-static-fields-and-data-slots to learn more. The documentation also suggests the following:

> *You can use the System.Threading.ThreadLocal<T> class to create thread-local objects that are initialized lazily when the object is first consumed.*

To illustrate a simple example, let me show you an example using the `ThreadLocal<T>` class. See the following program where I create two different tasks and each task receives the same initial value before they make some changes to it (notice that I also wait for these tasks to complete their executions before closing the application):

**POINT TO NOTE**

If you are reading about the ThreadLocal<T> class for the first time, I recommend you visit https://learn.microsoft.com/en-us/dotnet/api/system.threading.threadlocal-1?view=net-8.0 to see the different constructors and parameters available in this class. In this program, I have used the constructor that accepts a Func<T> parameter. In addition, I used the Value property to get and set the value of the instance of the current thread.

```
using static System.Console;
WriteLine("Understanding the ThreadLocal<T> class.");

var flag = new ThreadLocal<int>(() => 50);

var task1 = Task.Run(
    () =>
        {
          WriteLine($" Task1 received the value:{flag.Value}");
          flag.Value += 25;
          WriteLine($" Task1 changed the value to {flag.Value}");
        }
    );

var task2 = Task.Run(
    () =>
        {
          WriteLine($"\tTask2 received the value:{flag.Value}");
          flag.Value += 50;
          WriteLine($"\tTask2 changed the value to {flag.Value}");
        }
    );
```

```
Task.WaitAll(task1, task2);

WriteLine($"Before closing the application, the main thread
 contains the value: {flag.Value}");
```

Run the program now. Here is some sample output:

```
Understanding the ThreadLocal<T> class.
 Task1 received the value: 50
 Task1 changed the value to 75
        Task2 received the value: 50
        Task2 changed the value to 100
Before closing the application, the main thread contains the
  value: 50
```

You can see that each task received 50 as the initial value, and later, they changed this value inside them. However, those changes did not impact this value inside the main thread.

---

**Note** The `Q&A_UnderstandingThreadLocal` project contains this program. You can download it from the Apress website.

---

**Q4.7 In Demonstration 8, you initialized the thread-local variable to 1 by using the expression `() => 1`. Since you worked with an `int`, it was OK. I am curious to know how to initialize when you work with a reference type or user-defined value type.**

It is similar. You can write something like `()=> UserDefinedClass`.

# Reviewing Different Coding Styles

Congratulations! You are about to finish another useful chapter. Before you move into the next chapter, I'd like to draw your attention to a few coding styles. I have included this section to avoid any confusion if you see similar code in the future.

Throughout this book, to generate numbers, I used the `Enumerable` class. For example, to generate the numbers from 1 to 10, I used the following syntax:

```
List<int> numbers = Enumerable.Range(1, 10).ToList();
```

Or this:

```
var numbers = Enumerable.Range(1, 10).ToList();
```

In the first case, I explicitly mentioned the `type(List<int>)`. In the second case, I let the compiler infer the type. Both are fine. But as you progress more on the topic and learn PLINQ (or use LINQ), you'll understand that LINQ or PLINQ follows functional programming, which is nothing but declarative programming. So, if you are biased toward functional/declarative programming, you may prefer to use `var` instead of explicitly mentioning the type. I leave this preference to you.

**In this context, note that C# 12 (available with .NET 8) was released in November 2023 with some exciting features, and one of them is collection expressions**. Using this new feature, I showed you some alternative code in Chapter 1 ( you can refer to Demonstration 4 and Exercise 1.6 in that chapter) as well.

Let's take a quick look at the different approaches that show equivalent code.

**Approach 1:**

```
List<int> numbers = Enumerable.Range(1, 10).ToList();
```

**Approach 2:**

```
var numbers = Enumerable.Range(1, 10).ToList();
```

**Approach 3:**

```
List<int> numbers = [.. Enumerable.Range(1, 10)];
```

**But the following line will cause a compile-time error:**

```
var numbers4 = [.. Enumerable.Range(1, 10).ToList()];
```

I often follow Approach 1 and Approach 2 to support backward compatibility. But to make you familiar with the new syntax, in some cases, I have used Approach 3 as well.

## Introducing the ParallelEnumerable Class

There is a class called `ParallelEnumerable` that is the parallel equivalent of the `Enumerable` class. So, you can write something like the following as well:

**Approach 4:**

```
List<int> numbers = ParallelEnumerable.Range(1, 10).ToList();
```

Interestingly, if you use the previous line of code, Visual Studio will suggest you prefer Approach 3 and write something like the following:

**Approach 5:**

```
List<int> numbers = [.. ParallelEnumerable.Range(1, 10)];
```

You will see many useful methods inside this class for querying objects that implement `ParallelQuery{TSource}`. You'll see me using this class when I discuss parallel LINQ (PLINQ) queries in Chapter 5.

**POINT TO NOTE**

The `Enumerable` class provides methods for querying objects that implement `IEnumerable<T>`. The `ParallelEnumerable` class provides methods for querying objects that implement `ParallelQuery{TSource}`.

If you're interested, you can learn more about the class by referring to the documentation: https://learn.microsoft.com/en-us/dotnet/api/system.linq.parallelenumerable?view=net-8.0.

# Exercises

Check your understanding by attempting the following exercises:

**REMINDER**

As mentioned, for all the code examples, the Implicit Global Usings setting was enabled in Visual Studio. This is why you will not see me mentioning some other namespaces that were available by default. You can safely assume that all other necessary namespaces are available for these code segments in this exercise. The same comment applies to all exercises in this book.

**E4.1** True or False?

a) A parallel loop is always faster than the corresponding sequential loop.

b) The methods of the `Parallel` class improve CPU-bound tasks but not the I/O-bound tasks.

c) Thread-local variables can be used to fine-tune a parallel loop.

d) While instantiating the `ParallelOptions` class, you can set `MaxDegreeOfParallelism` to -1.

**E4.2** The following program prints the position of the characters of the string "abcde":

```
using static System.Console;

string input = "abcde";
int position = 0;
foreach (char c in input)
{
    WriteLine($"{c} is in position:{position++}");
}
```

Can you write an equivalent program using `Parallel.ForEach`?

(**Hint:** You can use the following overload of the `ForEach` method: `public static ParallelLoopResult ForEach<TSource>(IEnumerable <TSource> source, Action<TSource, ParallelLoopState, long> body);`)

**E4.3** Can you give an example where you'd like to use the `Stop` or `Break` method? These methods are very similar. Can you give an example where you'd prefer one over the other?

**E4.4** Can you predict the output of the following program?

```
using static System.Console;

try
{
    Action greet = new(() =>
    {
        WriteLine($"Hello reader! Started: {Task.CurrentId}");
```

```
        Thread.Sleep(500);
        WriteLine($"Have a nice day! Finished: {Task.
         CurrentId}");
    }
    );

    Action printMsg = new(() =>
    {
        WriteLine($"Trying to print a message. Started: {Task.
         CurrentId}");
        throw new InvalidOperationException("invalid
         operation");
    });

    Action sayBusy= new(() =>
    {
        WriteLine($"I'm performing a lengthy operation.
         Started: {Task.CurrentId}");
        Thread.Sleep(1500);
        WriteLine($"I am free now! Finished: {Task.CurrentId}");
    });

    Parallel.Invoke(greet, printMsg,sayBusy);
}
catch(AggregateException ae)
{
    foreach(Exception e in ae.InnerExceptions)
    {
        WriteLine($"Error: {e.Message}");
    }
}
```

**E4.5** Can you predict the output of the following program?

```
using static System.Console;

try
{
    ParallelOptions parallelOptions = new()
    {
      MaxDegreeOfParallelism = 0
    };
    List<int> numbers = [.. ParallelEnumerable.Range(1, 10)];
    var repeatedTask = Task.Run(() =>
    {
        Parallel.ForEach(
           numbers,
           parallelOptions,
            i =>
            {
                WriteLine($"Processed the number: {i}");
            });
    });
    repeatedTask.Wait();
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        WriteLine($"Error: {e.Message}");
    }
}
```

```
catch (ArgumentOutOfRangeException e)
{
    WriteLine($"Caught error: {e.Message}");
}
```

**E4.6** In this chapter, you saw me calculating the factorial of 10 using a parallel `for` loop. This time I want you to calculate the sum of the numbers from 1 to 1000. However, I want you to do this exercise using a parallel `foreach` loop. If possible, once you complete this exercise, fine-tune your application and compare the output as well.

# Summary

This chapter started with a quick recap of sequential loops and then focused on their parallel equivalents. It also analyzed some case studies to evaluate whether a parallel loop is beneficial compared to its counterpart (i.e., a sequential loop). In brief, it answered the following questions:

- How can you execute parallel loops in C#?
- How do you use `Parallel.ForEach`, `Parallel.For`, and `Parallel.Invoke` in an application?
- How can you manage a parallel loop using the `ParallelLoopState` and `ParallelOptions` parameters?
- How can you handle exceptions and manage cancellations while executing a parallel loop?
- How can you fine-tune a parallel loop using thread-local data?

# Solutions to Exercises

Here are the solutions for the exercises in this chapter.

## E4.1

The answers are as follows:

a) A parallel loop is always faster than the corresponding sequential loop. (**False**)

b) The methods of the `Parallel` class improve CPU-bound tasks but not I/O-bound tasks. (**True**)

c) Thread-local variables can be used to fine-tune a parallel loop. (**True**)

d) While instantiating the `ParallelOptions` class, you can set `MaxDegreeOfParallelism` to -1. (**True**)

## E4.2

The following program shows the sequential as well as the parallel version:

```
using static System.Console;

#region The sequential version
string input = "abcde";
int position = 0;
foreach (char c in input)
{
    WriteLine($"{c} is in position:{position++}");
}
#endregion
```

```
WriteLine("========");

#region The parallel version

Parallel.ForEach
 (
  input,
  (char c, ParallelLoopState state, long position) =>
   {
     WriteLine($"{c} is in position:{position++}");
    }
  );
#endregion
```

Here is some sample output for your reference:

```
a is in position:0
b is in position:1
c is in position:2
d is in position:3
e is in position:4
========
b is in position:1
c is in position:2
d is in position:3
a is in position:0
e is in position:4
```

## E4.3

You often see their presence in search-based algorithms.

Yes, they are indeed very similar. Let's see what the official documentation says about them. In the case of the `Stop` method, Microsoft

states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallelloopstate.stop?view=net-8.0):

> *Communicates that the Parallel loop should cease execution at the system's earliest convenience.*

However, in the case of the `Break` method, Microsoft states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallelloopstate.break?view=net-8.0#system-threading-tasks-parallelloopstate-break):

> *Communicates that the Parallel loop should cease execution* ***of iterations beyond the current iteration*** *at the system's earliest convenience.*

Now consider the case where you search for a particular element in a data structure using a parallel `for` loop. Typically, you can use either the `Stop` method or the `Break` method. However, in a search-based algorithm, if you find the required element, you can immediately stop the search operation.

Consider another case where you discovered a typical fault and, as a result, continuing the application does not make sense. So, in such cases, I'd like to use the `Stop` method.

## E4.4

Here is some sample output:

```
Trying to print a message. Started: 1
I'm performing a lengthy operation. Started: 2
Hello reader! Started: 3
Have a nice day! Finished: 3
I am free now! Finished: 2
Error: invalid operation
```

**Explanation:** Notice that you are seeing the exception at the end of the output. The documentation confirms the following (see https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff963549(v=pandp.10)?redirectedfrom=MSDN):

> *Any exceptions that occur during the execution of Parallel.Invoke are **deferred and rethrown when all tasks finish**. All exceptions are rethrown as inner exceptions of an AggregateException instance.*

## E4.5

You should see the following output:

```
Caught error: MaxDegreeOfParallelism ('0') must be a non-zero
value. (Parameter 'MaxDegreeOfParallelism')
Actual value was 0.
```

**Additional Note:** You are not allowed to set the value of the `MaxDegreeOfParallelism` property to zero or to a value that is less than -1; you'll see the `ArgumentOutOfRangeException`. However, you can set the value to -1. The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.paralleloptions.maxdegreeofparallelism?view=net-8.0#system-threading-tasks-paralleloptions-maxdegreeofparallelism):

> *A positive property value limits the number of concurrent operations to the set value. If it is -1, there is no limit on the number of concurrently running operations (with the exception of the ForEachAsync method, where -1 means ProcessorCount).*

# E4.6

Here is a sample implementation and its fine-tuned version:

```
using System.Diagnostics;
using static System.Console;

List<int> numbers = [.. ParallelEnumerable.Range(1, 1000)];

WriteLine($"Calculating the sum of 1 to 1000.");
#region Without using thread local variable
var stopWatch = Stopwatch.StartNew();
int total = 0;
Parallel.ForEach(
    numbers,
    (x) =>
    {
        Interlocked.Add(ref total, x);
    }
);
stopWatch.Stop();
WriteLine($"The sum is: {total}");
WriteLine($"The elapsed time is {stopWatch.ElapsedTicks} timer
  ticks.");
WriteLine("_____________");
#endregion

WriteLine($"Fine-tuning the application now.");

#region Using thread local variable
stopWatch = Stopwatch.StartNew();
total = 0;
Parallel.ForEach(
    numbers,
```

```
    () =>0,
    (int num, ParallelLoopState state, int subTotal) =>
    {
        return num + subTotal;
    },
   subTotal =>
   {
       Interlocked.Add(ref total, subTotal);
   }
 );
stopWatch.Stop();
WriteLine($"The sum is: {total}");
WriteLine($"Now the elapsed time is {stopWatch.ElapsedTicks}
 timer ticks.");

#endregion
```

Here is some sample output:

```
Calculating the sum of 1 to 1000.
The sum is:  500500
The elapsed time is 258695 timer ticks.

_____________
Fine-tuning the application now.
The sum is:  500500
Now the elapsed time is 31290 timer ticks.
```

CHAPTER 5

# Parallel LINQ

Language-Integrated Query (LINQ) was an important addition to .NET. Parallel LINQ (PLINQ) is its parallel implementation, which is an integral part of parallel programming. This chapter attempts to simplify the topic for you.

## Prerequisite Knowledge

In this section, I'll demystify some important characteristics of PLINQ. Careful study of this section will help you understand the topic better.

## Imperative vs. Declarative Programming

It is useful for PLINQ users to understand the difference between the imperative and declarative styles of coding. If you already know about them, you can skip this section. Otherwise, keep reading.

The imperative style of programming (also known as algorithmic programming) describes *how* to solve the problem by defining a sequence of commands or operations. This style of programming is characterized by explicit control flow and mutable state. For example, one may suggest: "Do this step, then do that step, and then get the result." Imperative programming is a step-by-step approach. Object-oriented programming languages such as C#, C++, Visual Basic, and Java are designed primarily to support imperative programming. This is why I assume you are familiar with this coding style.

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5_5

By contrast, declarative programming focuses on the result without trying to control the flow of the program. Instead of describing how to get the result, developers focus on *what* needs to be done. More specifically, in declarative programming, developers focus on defining the desired outcome instead of specifying step-by-step instructions and let the language or framework handle the implementation details. This is why declarative programming is often associated with higher-level abstractions, such as functions or expressions, and it encourages immutability and data-driven approaches. Languages like SQL, HTML/CSS, and functional programming languages like Haskell and F# exemplify declarative programming

The good news is that though C# is primarily object-oriented, it supports functional programming as well. It has many functional features, and some of them are common as well. To follow declarative programming, functional programmers use expressions, not statements. If you are familiar with LINQ, you have seen this before. Since functional programming is merely a form of declarative programming. you can say that C# supports declarative programming as well.

## PLINQ Supports Declarative Programming

Now if you need to exercise parallel programming, probably you would like to partition the work into multiple tasks, execute those tasks on different threads, and then merge the results. The good news is that all this is done by the underlying framework that PLINQ uses. This is why it is declarative, not imperative.

## Understanding LINQ Is Beneficial

The Microsoft documentation states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq):

> *PLINQ implements the full set of LINQ standard query operators as extension methods for the System.Linq namespace and has additional operators for parallel operations. PLINQ combines the simplicity and readability of LINQ syntax with the power of parallel programming.*

So, LINQ can help you learn and use PLINQ easily. Let me elaborate:

- LINQ follows the functional style of coding. It is declarative in nature, operates on an in-memory `IEnumerable` or `IEnumerable<T>` data source, and supports deferred execution. **PLINQ continues this legacy: it does the same.**
- Then how does the LINQ and PLINQ differ? PLINQ does the necessary things on your behalf to achieve parallelism: it partitions the data source into segments and executes queries on each segment on separate threads in parallel using multiple processors. This is why particular parallel queries can be significantly faster than the corresponding sequential queries.

# The Path Toward Parallelism

If you are familiar with LINQ queries, then you can impose parallelism on them easily. How? You simply use the extension method `AsParallel` as the entry point for PLINQ.

To illustrate, let's consider the following code. Given a list of numbers, this code picks numbers that are divisible by 5:

```
using static System.Console;
const int count = 30;
var numbers = Enumerable.Range(0, count);

// Using LINQ
var divisibleByFive=
    numbers
   .Where(x => x % 5 == 0);

WriteLine("\nNumbers divisible by 5 are:");
foreach(var x in divisibleByFive)
{
    Write($"{x}\t");
}
```

It is not a surprise that this code can produce the following output:

```
Numbers divisible by 5 are:
0       5       10      15      20      25
```

## Converting LINQ into PLINQ Using AsParallel

If you want to use a PLINQ query, you can use the `AsParallel` extension method in the previous LINQ query as follows:

```
var divisibleByFive=
    numbers
   .AsParallel()
   .Where(x => x % 5 == 0);
// There is no other change in the previous program
```

Upon executing, this modified code can produce different output. Here is a sample:

```
Numbers divisible by 5 are:
0       10      20      25      5       15
```

---

**Note** You can safely assume that the output will vary on your computer.

---

## Using the ParallelEnumerable Class

Since I used the `Enumerable` class in the previous program, I used `AsParallel()` to make it parallel. Since the `ParallelEnumerable` class is the parallel equivalent of `Enumerable`, I could start with the `ParallelEnumerable` class as well. For example, I could write something like the following:

```
numbers = ParallelEnumerable
          .Range(0, 30)
          .Where(x => x % 5 == 0);

// There is no other change in the previous program
```

Upon executing this query, my computer produces the following output:

```
Numbers divisible by 5 are:
0       5       10      20      25      15
```

Upon executing similar code, you may expect to see unordered output in a parallel environment. This is OK. However, you must note that **sequential output does not specify that the corresponding code did not execute in parallel. In addition, it is also possible that a PLINQ query**

**followed sequential execution**. Why? Microsoft states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq#parallel-vs-sequential-queries):

> *If PLINQ has a choice between a potentially expensive parallel algorithm or an inexpensive sequential algorithm, it chooses the sequential algorithm by default.*

So, a computer can produce ordered output as well as unordered output for a query. However, if you are sure that running a query in parallel mode is beneficial, you can instruct PLINQ to force parallelism by using `WithExecutionMethod`. PLINQ can also help you control the degree of parallelism. This is helpful when you work with I/O-intensive operations. I'll discuss this in a moment.

---

**Note** You can download the project `Demo_LinqToPlinq` to compare the different approaches and their corresponding results.

---

## Introducing ForAll

Notice that when you use `Enumerable.Range(0, count);`, the `Range` method returns an `IEnumerable<int>`. But when you use `ParallelEnumerable.Range(0, count);`, the `Range` method returns `ParallelQuery<int>`. You should keep this point in mind. In fact, if you investigate the following query:

```
divisibleByFive =
   numbers
  .AsParallel()
  .Where(x => x % 5 == 0);
```

you'll see that the `AsParallel` method also returns a `ParallelQuery<int>`, which is an important class. Refer to

https://learn.microsoft.com/en-us/dotnet/api/system.linq.parallelquery-1?view=net-8.0, which states that ParallelQuery<Tsource> represents a parallel sequence, and this class inherits from ParallelQuey and IEnumerable<TSource>. There you'll also find many useful methods and extension methods.

Among these methods, I often like to use the ForAll extension method in my code. Let me write a simple query in the following program:

```
using static System.Console;
const int count = 30;
WriteLine("\nNumbers divisible by 5 are:");
ParallelEnumerable
    .Range(0, count)
    .Where(x => x % 5 == 0)
    .ForAll(num=>Write($"{num}\t"));
```

Here is some sample output of this program:

```
Numbers divisible by 5 are:
0       10      25      5       20      15
```

**Author's Note:** When you download Demo_LinqToPlinq, you can see this code segment as well.

As a C# developer, you are familiar with foreach and for loops. You also learned about their parallel versions in Chapter 3. This time you saw the ForAll method and learned that this method also helps you invoke a specified action for each element of a source in parallel.

Now, you may wonder whether to use ForAll or Foreach or For. Let's refer to https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/potential-pitfalls-with-plinq#prefer-forall-to-foreach-when-it-is-possible to find Microsoft's thoughts on this. The documentation suggests the following:

> *Use the ForAll method to enable each thread to output its own results.*

## Q&A Session

**Q5.1 Why does Microsoft suggest using `ForAll` to `ForEach`?**

When you consume a parallel query result inside a `foreach` loop, the query result needs to be merged back into a thread and accessed serially by an enumerator. Though this step is essential for certain situations, it is not mandatory for every situation. So, you understand that by avoiding the merging step, the performance can be improved. `ForAll` does the same thing, but it does not require the merge step at the end.

This is why Microsoft says the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/potential-pitfalls-with-plinq):

> *The same issue applies to Parallel.ForEach. In other words, source.AsParallel().Where().**ForAll**(...) should be strongly preferred to Parallel.**ForEach**(source.AsParallel().Where(), ...).*

# Getting Familiar with PLINQ

Let's now explore PLINQ in more detail. You understand that though LINQ supports declarative programming, the execution is sequential. But when you deal with a lot of data, you may be interested in parallel execution. PLINQ is made for this. So, it's time for some more coding. Let's start.

## The First Program

In the following program, I generate some numbers using `Enumerable`'s `Range` method. After this, you will see a PLINQ query that checks whether a number is divisible by 5. If so, it increments the number by 100. This query will also show you the different task IDs that will be involved in the process. Finally, I display the numbers using a traditional `foreach` loop.

**Note** Later you will see me using the `ForAll` function instead of using a `foreach` loop or a `ForEach` function. Since this is just the beginning, let's learn slowly.

## Demonstration 1

Here is the complete program:

```
using static System.Console;

WriteLine("Introduction to PLINQ.");

var numbers = Enumerable.Range(1, 27);
// The query to get the new numbers.
var newNumbers =
    numbers
     .AsParallel()
     .Where(x => x % 5 == 0)
     .Select(num =>
     {
         WriteLine($" Processing: {num} ID:{Task.CurrentId}");
         return num + 100;
     });

WriteLine("The changed numbers are as follows:");
foreach (var number in newNumbers)
{
    Write($"{number}\t");
}
```

## Output

Here is some sample output:

```
Introduction to PLINQ.
The changed numbers are as follows:
 Processing: 15 ID:4
 Processing: 10 ID:3
 Processing: 25 ID:5
 Processing: 5 ID:2
 Processing: 20 ID:4
110     115     125     105     120
```

## Analysis

You can see that the PLINQ query processed the numbers in parallel. Notice that five numbers are updated and four different tasks (IDS 2,3,4 and 5) were involved in this process.

If you execute the program again, you may notice that these numbers appear in a different order. This is fine. It is because you cannot predict the processing order in advance.

## Q&A Session

**Q5.2 The last line in the previous output showed elements in an unordered fashion. I understand that in a parallel environment, this is an expected output. However, I am curious to know while using PLINQ, is it possible to control the order of the elements in the output?**

Yes. If you update the query using `AsOrdered()` as follows:

```
var newNumbers =
    numbers
     .AsParallel()
     .AsOrdered()
```

```
    .Where(x => x % 5 == 0)
    .Select(num =>
    {
        WriteLine($" Processing: {num} ID:{Task.CurrentId}");
        return num + 100;
    });
```

you can see that the output is as follows (notice that the elements are ordered in the final line):

```
// Previous lines of output are not shown to focus on this
// discussion
105     110     115     120     125
```

**Q5.3 What is the reason behind printing "The changed numbers as follows" text before "processing…." even though in code it is written on top?**

This is because you observed a deferred execution where we do not evaluate an expression until a realized value is required. Microsoft says the following (see https://learn.microsoft.com/en-us/dotnet/standard/linq/deferred-execution-lazy-evaluation):

> *Deferred execution can greatly improve performance when you have to manipulate large data collections, especially in programs that contain a series of chained queries or manipulations.*

However, certain query operators (and their corresponding functions for method syntax) promote immediate execution. In this context, Microsoft states the following:

> *Immediate execution means that the data source is read and the operation is performed once. All the standard query operators that return a scalar result execute immediately. You can force a query to execute immediately using the Enumerable.*

*ToList or Enumerable.ToArray methods.Immediate execution provides reuse of query results, not query declaration. The results are retrieved once, then stored for future use.*

So, if you want to perform an immediate execution, you can update the query where you append `ToList()` as follows:

```
var newNumbers =
    numbers
     .AsParallel()
     .Where(x => x % 5 == 0)
     .Select(num =>
     {
         WriteLine($" Processing: {num} ID:{Task.CurrentId}");
         return num + 100;
     })
     .ToList();
```

After this change, you can execute the program again to get output that looks like this:

```
Introduction to PLINQ.
Processing: 10 ID:3
 Processing: 25 ID:5
 Processing: 5 ID:2
 Processing: 15 ID:4
 Processing: 20 ID:4
The changed numbers are as follows:
105     110     115     120     125
```

Notice that to get the ordered output, I did not need to use the `AsOrdered` extension method as well.

## Do Parallel Queries Run Faster Than Sequential Queries?

The interesting point is that an attempt to make a parallel query does not guarantee that it will be faster than the corresponding sequential query. In fact, if you use an expensive partitioning strategy, it can significantly slow down the execution. So, it is important to understand the complexities of a query before you try to make it parallel.

To illustrate more on this, let me show you some sample programs. Go through the next three programs (Demonstration 2 to Demonstration 4).

## Demonstration 2

Here I measure the execution time for three equivalent code segments. Given some numbers, each part displays those numbers that are divisible by 5. The first code segment uses the traditional `foreach` loop, the next code segment uses one LINQ query, and the final one uses a PLINQ query as follows:

```
using System.Diagnostics;
using static System.Console;

var numbers = Enumerable.Range(0, 31);

Stopwatch sw;
sw = Stopwatch.StartNew();
WriteLine("Numbers divisible by 5 are (using foreach loop):");
foreach (int num in numbers)
{
    if (num % 5 == 0)
    {
        Write($"{num}\t");
    }
}
```

```
sw.Stop();
WriteLine($"Time taken:{sw.ElapsedMilliseconds} ms");

sw = Stopwatch.StartNew();

WriteLine("\nNumbers divisible by 5 are (using LINQ and the
 ForEach function):");
numbers
    .Where(num => num % 5 == 0)
    .ToList()
    .ForEach(num => Write($"{num}\t"));

sw.Stop();
WriteLine($"Time taken:{sw.ElapsedMilliseconds} ms");

sw = Stopwatch.StartNew();

WriteLine("\nNumbers divisible by 5 are (using PLINQ and the
 ForAll function):");
numbers
    .AsParallel()
    .Where(num => num % 5 == 0)
    .ForAll(num => Write($"{num}\t"));

sw.Stop();
WriteLine($"Time taken:{sw.ElapsedMilliseconds} ms");
```

## Output

Most of the time, on my computer, the code block that used the PLINQ query took more time to execute compared to other code segments. Let me include one of those samples for your reference:

```
Numbers divisible by 5 are (using foreach loop):
0       5       10      15      20      25      30      Time
taken:12 ms

Numbers divisible by 5 are (using LINQ and the ForEach function):
0       5       10      15      20      25      30      Time
taken:3 ms

Numbers divisible by 5 are (using PLINQ and the ForAll function):
5       20      25      30      10      0       15      Time
taken:23 ms
```

## Analysis

You can easily assume that it was due to the overhead associated with parallel processing. However, since my data set was very small, I could not get much benefit when I opted for parallel processing.

**POINT TO NOTE**

I have measured the performance using `ElapsedMilliseconds`. The documentation at https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-measure-plinq-query-performance suggests that for more fine-grained measurement, you should use `ElapsedTicks` instead of `ElapsedMilliseconds`. It also states the following:

*To get a deeper and richer view of the interaction of the query threads with one another and with other running processes, use the Concurrency Visualizer.*

Our primary goal in this chapter is to understand the fundamentals of PLINQ. The discussion of the Concurrency Visualizer is beyond the scope of this book.

## Q&A Session

**Q5.4 In Demonstration 2, the PLINQ query used the `ForAll` function instead of `ForEach`. Is there any specific thought behind this?**

Since I did not need to merge the results at the end (see Q&A Session 5.1), following Microsoft's suggestion, I preferred `ForAll` over `ForEach`.

## Demonstration 3

Now consider the following program that uses a time-consuming method `GetResult` in LINQ:

```
using System.Diagnostics;
using static System.Console;

Stopwatch sw = Stopwatch.StartNew();
var numbers = Enumerable.Range(0, 4);

var results =
    numbers
    .Select(x => GetResult(x + 1));

results
    .ToList()
    .ForEach(x => WriteLine(x + "\t"));

sw.Stop();
WriteLine($"Time taken: {sw.ElapsedMilliseconds} ms");

static int GetResult(int number)
{
    int waitTimeMs = number * 1000;
    // Simulating the delay
    Thread.Sleep(waitTimeMs);
    return waitTimeMs;
}
```

## Output

You can easily predict that the execution time will not be less than (1000+2000+3000+4000) ms. I have included some sample output for you:

```
1000
2000
3000
4000
Time taken: 10098 ms
```

## Enabling Parallelization

If you are using a multicore machine, an equivalent PLINQ query can improve the performance. So, let's enable parallelization by updating the previous query as follows:

```
// There is no change in the previous code
var results =
    numbers
    .AsParallel()
    .Select(x => GetResult(x + 1));
// There is no change in the remaining code
```

## Output

Execute the program one more time. Here is some sample output:

```
1000
2000
3000
4000
Time taken: 4075 ms
```

## Analysis

You can see that the program is almost 2.5 times faster now. The time difference between the approaches will increase if you keep increasing the numbers.

**Demonstrations 2 and 3 also show that certain PLINQ queries can improve the performance of a program, but the same is not true for all PLINQ queries.**

---

**Note** In this context, I encourage you to read Igor Ostrovsky's thoughts at https://devblogs.microsoft.com/pfxteam/plinq-queries-that-run-sequentially/.

---

## Q&A Session

**Q5.5 In Demonstration 3, you have used the following code segment:**

```
var results =
    numbers
    .Select(x => GetResult(x+1));

results
    .ToList()
    .ForEach(x => WriteLine(x + "\t"));
```

**But it appears to me that you could use a single query as follows:**

```
numbers
.Select(x => GetResult(x + 1))
.ToList()
.ForEach(x => WriteLine(x + "\t"));
```

**Is there any specific reason behind this?**

Nice observation. The `ToList` method forces an immediate execution, but I opted for a deferred execution.

**Q5.6 In Demonstration 3, you have the `ForEach` function instead of a `foreach` loop. Do you think `ForEach` is better than `foreach`?**

No. In fact, `foreach` is probably more readable than `ForEach`. But I have found `ForEach` helpful in certain scenarios because of the following reasons:

- You can reduce the code length.
- Once you start composing functions using method chaining, you'll find it is more readable at the end of a long chain. In fact, in the case of parallel programming, you may notice a similar coding style.

In Demonstration 3, since you're already converting the results to a list with `ToList()`, using `ForEach` is a reasonable choice. However, if you didn't need to convert to a list and wanted more control over the loop, `foreach` might be a better choice.

---

**Note** In Demonstration 3, I have not changed the `Foreach` function only for an easy comparison between a LINQ query and its equivalent PLINQ query. However, following Microsoft's suggestion, I'd recommend using the `ForAll` function instead of the `ForEach` function whenever possible, particularly to enable each thread to output its own results.

---

**Q5.7 The `ForEach` function is defined in the `List<T>` class only. Similar support is not available for other data structures. So, it appears to me that its usage is limited. Is this correct?**

You can define an extension method to provide similar support for other data structures as well. If interested, you can download the project `Demo_TraversingDictionaryUsingForEach` where you see a sample implementation.

**Author's Note:** I showed this program in my other book *Introducing Functional Programming Using C#*, also published by Apress.

# Forcing Parallelism and Controlling the Degree

As mentioned, in some situations, PLINQ can run the query sequentially. However, if you are sure that running a query in parallel mode is beneficial, you can instruct PLINQ to force parallelism. PLINQ can also help you control the degree of parallelism. This is helpful when you work with I/O-intensive operations. Let's examine them.

## Demonstration 4

In Demonstration 2, we analyzed the performance of three equivalent code segments. Let's append one more code segment in that demonstration. You may note that using this code, I try to force parallelism as well as control the degree of parallelism using two useful methods, called `WithExecutionMode` and `WithDegreeOfParallelism`. Here I'll append the following code segment in Demonstration 2 and measure the execution time one more time:

```
// The previous code is shown in demonstration 2

sw = Stopwatch.StartNew();
WriteLine("\nNumbers divisible by 5 are(using PLINQ(dop-4):");
numbers
    .AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .WithDegreeOfParallelism(4)
    .Where(num => num % 5 == 0)
```

```
    .ForAll(num => Write($"{num}\t"));
sw.Stop();
WriteLine($"Time taken:{sw.ElapsedMilliseconds} ms");
```

## Output

My computer shows a significant performance gain for the added code segment. Here is some sample output (see the bold portion):

```
Numbers divisible by 5 are (using foreach loop):
0       5       10      15      20      25      30   Time taken:14 ms

Numbers divisible by 5 are (using LINQ and the ForEach function):
0       5       10      15      20      25      30   Time taken:1 ms

Numbers divisible by 5 are (using PLINQ and the ForAll function):
10      15      25      30      0       5       20   Time taken:22 ms

Numbers divisible by 5 are(using PLINQ(dop-4):
25      20      30      10      15      0       5    Time taken:1 ms
```

**POINT TO NOTE**

The documentation suggests the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq#parallel-vs-sequential-queries):

In cases where a query is performing a significant amount of non-compute-bound work such as File I/O, it might be beneficial to specify a degree of parallelism greater than the number of cores on the machine.

I assume that careful study of this section gives you fundamental ideas about PLINQ. It's time to explore more about this topic. Continue reading.

# Merging Data

In a concurrent environment, you will deal with multiple threads. Exercising parallel queries, you can divide complex work into multiple parts and execute those parts in different threads. Upon completion, you may want to merge the results. Briefly, these are the steps:

- Partition the data into multiple parts.
- Multiple threads work on these parts and produce the corresponding results.
- Finally, collect these results and combine (aka merge) them. In other words, consumers consume the results that are produced by these threads.

The last step is common but not mandatory for every situation. Typically, it depends on the operations that you perform on the application.

For example, you may want to consume these results on a particular thread using a `foreach` loop. This case requires you to merge these results before you exercise the loop. But, if displaying the output is the only concern, then `ForAll` can be a better choice.

## Merge Options

Now the question is **if needed, how do you merge the data?** The answer to this question can lead us to various approaches such as the following:

- Buffer all the data before you merge. Then pass the whole set of data to the consuming thread.
- Buffer the data chunk by chunk and yield those chunks periodically to the consuming thread.

- Yield the elements as soon as they are available. Notice that in this approach you do not need to maintain any buffer before you send the data to the consuming thread. This is analogous to "streaming" the data.

To exercise these options, you can use the `WithMergeOptions` method. This is an extension method that accepts `ParallelMergeOptions` as a parameter. On further investigation, you can see that `ParallelMergeOption` is an enumeration type and has four members: `Default`, `NotBuffered`, `AutoBuffered`, and `FullyBuffered`. If you expand the `Default` member, you can see that the default merge type is auto-buffered. Figure 5-1 shows a sample screenshot for reference.

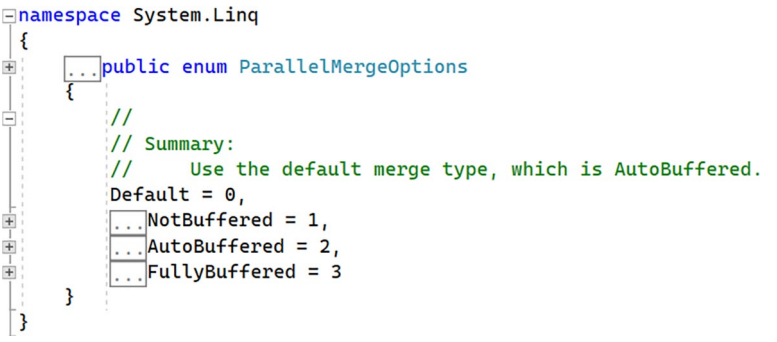



***Figure 5-1.*** *The members of ParallelMergeOptions*

## Demonstration 5

Let's see a demonstration. The goal of the demonstration is to exercise different merge options and investigate the results.

Using a PLINQ query, the following program produces some data. While producing the data, it uses the `FullyBuffered` option. Here you will see the following query:

```
var incrementedvalues =
    numbers
    .AsParallel()
```

```
    .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
    .Select(x =>
    {
        var temp = Math.Pow(x,2);
        Write($"[{temp}]");
        return temp;
    });
```

You can see that I have used a square bracket ([]) to mark the data being produced by the program.

Later, you'll see a foreach loop to consume this data. It is as follows:

```
foreach (var number in incrementedvalues)
{
    Write($"{number} \t");
}
```

You understand that this is a simple demonstration that follows a parallel producer-consumer pattern. Let's see the complete program:

```
using static System.Console;

List<int> numbers = Enumerable.Range(1, 25).ToList();

WriteLine("Making square.");
var incrementedvalues =
    numbers
    .AsParallel()
    .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
    .Select(x =>
    {
        var temp = Math.Pow(x,2);
        Write($"[{temp}]");
```

```
        return temp;
    });

foreach (var number in incrementedvalues)
{
    Write($"{number} \t");
}
```

## Output

Here is some sample output:

```
Making square.
[196][1][400][64][81][100][121][144][225][441][4][256][289]
[324][361][169][484][9][529][16][25][576][625][36][49]1     4
9       16      25      36      49      64      81
100     121     144     169     196     225     256    289
324     361     400     441     484     529     576     625
```

The output shows that after producing all the data, the program started consuming that data. Now change the option to `NotBuffered`. So, here is the updated query:

```
var incrementedvalues =
    numbers
    .AsParallel()
    .WithMergeOptions(ParallelMergeOptions.NotBuffered)
    .Select(x =>
    {
        var temp = Math.Pow(x,2);
        Write($"[{temp}]");
        return temp;
    });
```

Here is some sample output for this change:

```
Making square.
[400][1][196][64][441][484][225][81][4][529][576][100][9]
[16]1  [121][256][289][324][361][144][625][25]64       [169]
[36]196    400     [49]4   81      225     441     9
100     256     484     16      121     289     529     25
144      324     576     36      169     361     625     49
```

The output shows that the program started consuming the data (for example, see 1 came before [121], 64 came before [169], and so on) before all the data was produced.

## Suggestions

Before I finish this section, I'd like you to notice the operators that you use in a query. These are good indicators of what kind of buffer is required. For example, `Reverse` or `OrderBy` cannot yield any element until the complete data is generated. On the contrary, `ForAll` can yield the elements immediately. It does not require any buffer.

## Q&A Session

**Q5.8 I understand that in some special queries, I may need to generate the complete data before yielding them. In those cases, I need to use the `FullyBuffered` option. But what happens if I choose a different option? For example, I'd like to know instead of choosing the `FullyBuffered` option what happens if I choose a different option such as `NotBuffered`. Will it cause a compile-time error?**

No. In those cases, the request will be ignored. Here is an example. Replace the PLINQ query in the previous program with the following one (see the change in bold):

```
var orderedList =
     numbers
    .AsParallel()
    .WithMergeOptions(ParallelMergeOptions.NotBuffered)
    .Select(x =>
    {
        var temp = Math.Pow(x, 2);
        Write($"[{temp}]");
        return temp;
    })
    .OrderDescending();
```

Here is some sample output after this change:

```
[1][196][225][256][289][400][441][484][529][576][625]
[324][361][4][9][16][25][64][36][49][81][100][121][144]
[169]625   576      529     484     441     400     361
324     289     256     225     196     169     144     121
100    81       64      49      36      25      16      9
4       1
```

You can see that the data consumption started only after all the data was produced.

Note that since PLINQ is intelligent enough, you do not need to specify the merge options in most of these cases. But through testing and experience, you may decide that in certain cases, opting for a nondefault option is beneficial. For example, to make the user interface more responsive, you may opt for those nondefault options.

# Managing Special Scenarios

In Chapter 2, you learned about handling exceptions and managing cancellations in task programming. Those concepts are applicable when you use PLINQ as well. Let's look at some case studies.

## Handling Exceptions

Read the following points to understand the upcoming program:

- At first, the program generates some numbers.
- Once that is done, it uses a PLINQ query that performs multiple things.
- First, it filters the numbers that are divisible by 10.
- Then, it calculates a square of those numbers. For example, once it finds the number 10, it will make it 10*10=100.
- The final part of the query displays the calculated value in the console.
- To display the calculated values, I use the `ForAll` method because I do not need to order the output.
- I create an exceptional situation whenever the transformed number is greater than 6400. In this case, the program throws a custom exception. I named this exception `ExceedsCustomLimitException` for easy readability.

Now understand that since the PLINQ query raises an exception, I need to handle the situation. The following program handles the situation using the concept you learned about in Chapter 2. This is the reason you will see a `catch` block to handle `AggregateException` in this program too.

## Demonstration 6

Here is the complete program:

```
using static System.Console;
using CustomExceptions;

var numbers = Enumerable.Range(0, 100);
try
{

   numbers
   .AsParallel()
   .Where(x => x % 10 == 0)
   .Select(
     x =>
      {
        int temp = x * x;
        if (temp > 6400)
          throw new ExceedsCustomLimitException($"The
           calculated value {temp} exceeds 6400");
       WriteLine($"{x} is processed by task:[{Task.CurrentId}]");
      return temp;
    }
  )
  .ForAll(num => WriteLine($"The calculated number is: {num}"));
}

catch (AggregateException ae)
{
    // Approach-1
    foreach (Exception e in ae.InnerExceptions)
```

```
    {
        WriteLine($"Error Type: {e.GetType().Name}, Message:
         {e.Message}");
    }

    //// Approach-2
    // ae.Handle(e =>
    // {
    //     WriteLine($"Error Type:{e.GetType().Name},
    //     Message:{e.Message}");
    //     return true;
    // });
}

namespace CustomExceptions
{
    public class ExceedsCustomLimitException : Exception
    {
        public ExceedsCustomLimitException()
        {
            // Some other code, if any
        }

        public ExceedsCustomLimitException(string message)
            : base(message)
        {
            // Some other code, if any
        }

        public ExceedsCustomLimitException(string message,
         Exception inner): base(message, inner)
        {
```

```
            // Some other code, if any
        }
    }
}
```

## Output

Here is some sample output:

```
30 is processed by task:[3]
80 is processed by task:[5]
The calculated number is: 6400
50 is processed by task:[4]
The calculated number is: 2500
60 is processed by task:[4]
The calculated number is: 3600
0 is processed by task:[2]
The calculated number is: 900
70 is processed by task:[4]
40 is processed by task:[3]
The calculated number is: 1600
The calculated number is: 4900
The calculated number is: 0
10 is processed by task:[2]
The calculated number is: 100
20 is processed by task:[2]
The calculated number is: 400
Error Type: ExceedsCustomLimitException, Message: The
 calculated value 8100 exceeds 6400
```

## Handling Cancellations

You can manage cancellation operations as well. The approach is almost the same as you saw in Chapter 2. To show you a sample case study on cancellations while a PLINQ query is executing, I will modify the previous program (Demonstration 6) now.

Since this program intends to handle both exceptions and cancellations, I updated the logic a little bit as well. Let's review the important changes.

In Demonstration 6, whenever the calculated number was greater than 6400, I threw a custom exception. This time I do not check only the calculated number. Instead, I generate (and check) a random number as well. If this random number is an even number and the calculated value is greater than 6400, I throw the custom exception. However, if the random number is an odd number and the calculated value is greater than 6400, I trigger a cancellation. This is why you'll see the following code in the upcoming program:

```
if (temp > 6400)
{
 if (random % 2 == 0)
 {
  throw new ExceedsCustomLimitException($"The calculated value
   {temp} exceeds 6400");
 }
 else
 {
  cancellationTokenSource.Cancel();
 }
}
```

To handle a cancellation scenario, the program creates a CancellationToken instance. So, you see the following lines:

```
CancellationTokenSource cancellationTokenSource = new();
CancellationToken token = cancellationTokenSource.Token;
```

But the most important part is to understand how you associate the CancellationToken instance to the query. You will see the use of the extension method WithCancellation as follows:

```
numbers
 .AsParallel()
 .Where(x => x % 10 == 0)
 .WithCancellation(token)
 .Select(
   x =>
   {
    // Some code here
   }
  );
```

This time you may need a catch block to handle the OperationCanceledException. So, you'll also see the following block:

```
catch (OperationCanceledException oce)
{
    WriteLine($"Error:{oce.Message}");
}
```

You should not have a problem understanding the remaining part of the program. Let's look at it now.

## Demonstration 7

Here is the complete program. Notice the important changes in bold.

```
using static System.Console;
using CustomExceptions;

var numbers = Enumerable.Range(0, 100);
CancellationTokenSource cancellationTokenSource = new();
CancellationToken token = cancellationTokenSource.Token;
try
{
    numbers
     .AsParallel()
     .Where(x => x % 10 == 0)
     .WithCancellation(token)
     .Select(
         x =>
            {
                int temp = x * x;
                int random = new Random().Next(0, 2);
                if (temp > 6400)
                {
                    if (random % 2 == 0)
                    {
                        throw new ExceedsCustomLimit
                         Exception($"The calculated value {temp}
                         exceeds 6400");
                    }
                    else
                    {
                        cancellationTokenSource.Cancel();
                    }
                }
```

```
                WriteLine($"{x} is processed by task:[{Task.
                 CurrentId}]");
                return temp;
            }
     )
    .ForAll(num => WriteLine($"The calculated number is: {num}"));
}

catch (AggregateException ae)
{
    // Approach-1
    foreach (Exception e in ae.InnerExceptions)
    {
        WriteLine($"Error Type: {e.GetType().Name}, Message:
         {e.Message}");
    }

    //// Approach-2
    // ae.Handle(e =>
    // {
    //     WriteLine($"Error Type:{e.GetType().Name},
             Message:{e.Message}");
    //     return true;
    // });
}
catch (OperationCanceledException oce)
{
    WriteLine($"Error: {oce.Message}");
}

// The CustomExceptions namespace is the same. It is not shown
// again to avoid repetition.
```

**Note** You can download the project `Demo7_ManagingCancellations` from the Apress website to see the complete implementation. It is in the Chapter 5 folder.

## Output

Here is some sample output I got when a cancellation was triggered:

```
30 is processed by task:[3]
80 is processed by task:[5]
The calculated number is: 6400
50 is processed by task:[4]
The calculated number is: 2500
The calculated number is: 900
60 is processed by task:[4]
0 is processed by task:[2]
The calculated number is: 0
The calculated number is: 3600
70 is processed by task:[4]
The calculated number is: 4900
10 is processed by task:[2]
The calculated number is: 100
90 is processed by task:[5]
40 is processed by task:[3]
The calculated number is: 1600
The calculated number is: 8100
20 is processed by task:[2]
The calculated number is: 400
Error: The query has been canceled via the token supplied to
 WithCancellation.
```

**Author's Note:** Whenever the program raises the other exception (`ExceedsCustomLimitException`), you should expect to see similar output as shown in Demonstration 6. So, I did not present the output again.

## Q&A Session

**Q5.9 If I handle a cancellation in the user code, do I need to use `WithCancellation` in the query definition?**

No. However, Microsoft does not want you to avoid using this. The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-cancel-a-plinq-query):

> *When you handle the cancellation in user code, you do not have to use WithCancellation in the query definition. However, we recommend that you do use WithCancellation, because WithCancellation has no effect on query performance and it enables the cancellation to be handled by query operators and your user code.*

**MICROSOFT'S RECOMMENDATION**

I recommend that you visit https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-cancel-a-plinq-query. There you can find some additional information as well as some useful guidelines on this topic.

# Exercising Aggregation

Aggregation operations are common in programming. Using an aggregation function, you can compute a single value from a collection of values. LINQ queries are very efficient in such cases. Let me show you a few code segments that will support my claim.

**Example 1:**

Given the numbers 1 to 10, the following code creates the squares of each number and calculates the sum of those squares:

```
using static System.Console;
int sum = Enumerable
    .Range(1, 10)
    .Sum(x => x * x);
WriteLine($"The sum of squares:{sum}");
```

**Example 2:**

Given these numbers, you can also find the maximum number using the following code:

```
int max = Enumerable
    .Range(1, 10)
    .Max();
WriteLine($"Maximum:{max}");
```

In this example, if you replace the `Max` function with `Average`, you can calculate the average of these numbers as well.

**You may note that the methods `Sum`, `Max`, and `Average` are nothing but ready-made extension methods to help you perform aggregation operations.**

## Sequential Custom Aggregation

Instead of using specialized aggregation methods, you can use a general method called `Aggregate`. This method allows you to perform custom aggregation. Here is a sample that can also display the sum of the squares for the numbers 1 to 10 (I have chosen the initial accumulator value, or the seed value 0):

```
int total = Enumerable
    .Range(1, 10)
    .Select(x => x * x)
    .Aggregate(0, (subTotal, nxtNumber) => subTotal + nxtNumber);
WriteLine($"The sum of squares: {total}");
```

If you want to see the intermediate values, I can further modify the query as follows:

```
int total = Enumerable
    .Range(1, 10)
    .Select(x => x * x)
    .Aggregate(0,(subTotal,nxtNumber) =>
    {
        int temp = subTotal+ nxtNumber;
        WriteLine($"{subTotal}+{nxtNumber}={temp}");
        return temp;
    });
WriteLine($"The sum of squares: {total}");
```

This time you can see the following output, which is very clear:

```
0+1=1
1+4=5
5+9=14
14+16=30
30+25=55
55+36=91
91+49=140
140+64=204
204+81=285
285+100=385
The sum of squares: 385
```

Till now, you have seen some aggregation operations that perform sequential executions. **Now the question is: how can we perform custom aggregation in a parallel environment?** The following section answers this question.

# Parallel Custom Aggregation

Inside the `ParallelEnumerable` class, you'll see several overloads of the `Aggregate` method; see Figure 5-2.

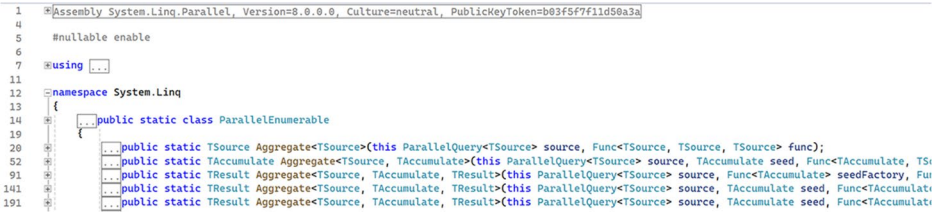



***Figure 5-2.*** *Overloads of aggregation methods*

The interesting point is that if you look into these methods, you'll find overloads that are not available in the sequential implementation. Let me pick a complex version that is particularly designed to work in a parallel environment. Here it is:

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
  this ParallelQuery<TSource> source,
  TAccumulate seed,
  Func<TAccumulate, TSource, TAccumulate> updateAccumulatorFunc,
    Func<TAccumulate, TAccumulate, TAccumulate> combineAccumulatorsFunc,
     Func<TAccumulate, TResult> resultSelector)
{
// Implementation is not shown here
```

Before I show you an application of this method, I suggest you read the method definition, which can be easily seen in Visual Studio. This description will help you understand the method. In brief, it works like this:

- You choose a seed value.
- You partition the data (source) into several subpartitions and apply an accumulator function on each element of these partitions.
- Each subpartition will yield elements. Another accumulator function will work on these yielded elements.
- The final accumulator value will be transformed into a result value.

In the upcoming program, I keep the supporting comments to help you match these steps.

## Demonstration 8

It is a time for another demonstration where you exercise custom aggregation in a parallel environment. The last time you calculated $\mathbf{1^2+2^2+ ...+10}$.[2] This time you compute the same in a parallel environment. Here is the complete program:

```
using static System.Console;
int parallelSum = ParallelEnumerable
    .Range(1, 10)
    .Select(x => x * x)
    .Aggregate(
        // initialize subtotal/ seed value.
        0,
        // Executes this on each thread
```

```
        (subTotal, nxtNumber) =>
        {
            int temp = subTotal + nxtNumber;
            WriteLine($"subTotal={subTotal}, next number=
             {nxtNumber}, temp={temp} id:[{Task.CurrentId}]");
            return temp;
        },
        // Aggregating the subtotals from all threads
        (total, subTotal) =>
        {
            int temp2 = total + subTotal;
            WriteLine($"total={total},subTotal={subTotal},temp2
             ={temp2}");
            return temp2;
        },
        // Processing the final result(if required) before
        // you return
        total => total
    );
WriteLine($"The sum of squares: {parallelSum}");
```

## Output

As always, the output will vary on your computer. Here is some sample output that was produced on my computer:

```
subTotal=0, next number=16,temp=16 id:[3]
subTotal=0, next number=81,temp=81 id:[5]
subTotal=0, next number=49,temp=49 id:[4]
subTotal=0, next number=1,temp=1 id:[2]
subTotal=16, next number=25,temp=41 id:[3]
```

```
subTotal=81, next number=100,temp=181 id:[5]
subTotal=49, next number=64,temp=113 id:[4]
subTotal=1, next number=4,temp=5 id:[2]
subTotal=41, next number=36,temp=77 id:[3]
subTotal=5, next number=9,temp=14 id:[2]
total=14,subTotal=77,temp2=91
total=91,subTotal=113,temp2=204
total=204,subTotal=181,temp2=385
The sum of squares: 385
```

## Analysis

Notice that four tasks (IDs 2, 3, 4, and 5) performed the computation as follows:

- Task ID 2 computes its final subtotal as 14 (see the last appearance of `id:[2]`).
- Task ID 3 computes its final subtotal as 77.
- Task ID 4 computes its final subtotal as 113.
- Task ID 5 computes its final subtotal as 181.

Later these subtotals are added to compute the final total.

## Q&A Session

**Q5.10 In the previous demonstration, you did not use the `AsParallel` method inside the query. Was that intentional?**

You already learned that the `ParallelEnumerable` class is the parallel equivalent of `Enumerable`. In addition, notice that `Enumerable.Range(1, 10)` returns `IEnumerable<int>`, but `Enumerable.Range(1, 10).AsParallel()` returns `ParallelQuery<int>`. Using `ParallelEnumerable.Range(1,10)`, you already got a `ParallelQuery<int>` as well. Since I used `ParallelEnumerable.Range` already, it was not needed for me to write

something like `ParallelEnumerable.Range (1,10).AsParallel()` inside the query. However, you may note that the program can still work despite this redundant call.

# Exercises

**E5.1** Pick the correct statements from the following set of statements:

a) A parallel query is always faster than a sequential query.

b) PLINQ follows declarative programming.

c) `WithDegreeOfParallelism` is an extension method that can be used to set the degree of parallelism in a parallel query.

d) The `ForAll` extension method does not require a merge step at the end.

e) If we use the `ForAll` method, the order of execution is deterministic.

f) If you specify a merge option such as `AutoBuffered` in a PLINQ query, the system may ignore the request, if necessary.

**E5.2** Can you write a program that uses a PLINQ query to find prime numbers between 1 and 100?

**E5.3** Can you predict the output of the following sequential query (notice that a few lines are commented here)?

```
using static System.Console;

int numberAbove20ComesFirst =
    Enumerable
    .Range(1, 50)
```

```
    // .AsParallel()
    // .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    // .WithDegreeOfParallelism(4)
    .Where(x => x % 2 == 0)
    .Aggregate
     (
       20,
       (tempHighest, next) =>
       {
           if (tempHighest > 20)
           {
               return tempHighest;
           }
           else
           {
               int temp = tempHighest >= next ?
                 tempHighest : next;
               WriteLine($"seed={tempHighest},
                 next={next},temp={temp}");
               return temp;
           }
       }
     );

WriteLine($"The number above 20 that appears first is:
 {numberAbove20ComesFirst}");
```

**E5.4** Uncomment the following portion in the previous program:

```
// .AsParallel()
// .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
// .WithDegreeOfParallelism(4)
```

Can you predict the output now?

**E5.5** Can you predict the output of the following program?

```
using static System.Console;

var numbers = Enumerable.Range(0, 100);
CancellationTokenSource cts = new();
CancellationToken token = cts.Token;

var cancellationTask = Task.Run(() =>
{
    // We'll trigger a cancellation request shortly
    Thread.Sleep(300);
    cts.Cancel();
});

var anotherTask = Task.Run(SomeMethod, token);
anotherTask.Wait();
WriteLine("End of the main thread.");

void SomeMethod()
{
    try
    {
         numbers
         .AsParallel()
         .Where(x => x % 10 == 0)
         .WithCancellation(token)
         .Select(
             x =>
             {
                 token.ThrowIfCancellationRequested();
                 Thread.Sleep(100);
                 return x;
```

```
            }
        )
         .ForAll(num=>WriteLine($"The processed number is:
          {num}"));
    }
    catch (AggregateException ae)
    {
        foreach (Exception e in ae.InnerExceptions)
        {
            WriteLine($"Error Type:{e.GetType().Name},
             Message:{e.Message}");
        }
    }
    catch (OperationCanceledException oce)
    {
        WriteLine($"Error:{oce.Message}");
    }
}
```

**E5.6** You have learned about parallel custom aggregation in this chapter. Can you calculate the factorial of 10 using the same concept?

# Summary

This chapter started with some prerequisite discussions that can help you understand PLINQ better. Then it showed you how to convert a sequential query into a parallel query. After this, it discussed different aspects of PLINQ. In brief, it answered the following questions:

- Why is PLINQ declarative?
- Why are some PLINQ queries faster than sequential queries (and vice versa)?

- How can you force parallelism in a query?
- How can you exercise different merge options before outputting the result?
- How can you handle exceptions, and how can you handle exceptions while exercising a PLINQ query?
- How can use apply aggregation in a parallel environment?

# Solutions to Exercises

Here are the solutions for the exercises in this chapter.

## E5.1

The answers are shown in bold as follows:

a) A parallel query is always faster than a sequential query. (**False**)

b) PLINQ follows declarative programming. (**True**)

c) `WithDegreeOfParallelism` is an extension method that can be used to set the degree of parallelism in a parallel query. (**True**)

d) The `ForAll` extension method does not require a merge step at the end. (**True**)

e) If we use the `ForAll` method, the order of execution is deterministic. (**False**)

f) If you specify a merge option such as `AutoBuffered` in a PLINQ query, the system may ignore the request, if necessary. (**True**)

# E5.2

Here is a sample program:

```
using static System.Console;

const int count = 100;
var numbers = Enumerable.Range(1, count);

var primes = numbers
   .AsParallel()
   .Where(x =>
   {
       if (x == 1) return false;
       if (x == 2) return true;
       var boundary = (int)Math.Floor(Math.Sqrt(x));
       for (int i = 2; i <= boundary; i++)
       {
           if (x % i == 0) return false;
       }
       return true;
   });
WriteLine("Prime numbers are:");
primes
    .ToList()
    .ForEach(primes => Write(primes + "\t"));
```

This program produces the following output:

```
Prime numbers are:
2   3   5   7   11  13   17   19  23   29   31   37   41
43   47   53   59   61   67   71   73   79   83   89   97
```

**Code explanation:** A number is prime if it is divisible by 1 and the number itself. For example, 41 is divisible by 1 and 41 only. This is why to check whether a number, say 41, is a prime number, you can loop through 1 to 41 and find that you have not received more than two factors. But instead of looping up to a given number (say n), mathematically, we can prove that testing up to the square root of n should be enough for us. This is why I started looping from 2 and testing up to the square root of the given count (i.e., 100).

# E5.3

The output of this program is predictable. The final value will always be 22. The following output clearly shows that the control followed the sequential execution:

```
seed=20, next=2,temp=20
seed=20, next=4,temp=20
seed=20, next=6,temp=20
seed=20, next=8,temp=20
seed=20, next=10,temp=20
seed=20, next=12,temp=20
seed=20, next=14,temp=20
seed=20, next=16,temp=20
seed=20, next=18,temp=20
seed=20, next=20,temp=20
seed=20, next=22,temp=22
The number above 20 that appears first is: 22
```

## E5.4

After this modification, the output of the program is unpredictable. Here is some sample output that appears on my computer:

```
seed=20, next=14,temp=20
seed=20, next=28,temp=28
```

The number that is greater than 20 that appears first is 28.

However, remember that since we are exercising a parallel query without preserving the order, the numbers can be processed in random order. For example, when I executed the same program online at SharpLab (https://sharplab.io), I saw the following output:

```
seed=20, next=2,temp=20
seed=20, next=4,temp=20
seed=20, next=6,temp=20
seed=20, next=8,temp=20
seed=20, next=10,temp=20
seed=20, next=12,temp=20
seed=20, next=40,temp=40
The number above 20 that appears first is: 40
```

## E5.5

Once again, there is no doubt that the output can vary. But I want you to notice how a cancellation request is handled in this code. You may note that I imposed some forced delay so that the cancellation fires while the query is executing. Here is some sample output on my computer:

```
The processed number is: 0
The processed number is: 80
The processed number is: 30
The processed number is: 90
```

```
The processed number is: 40
The processed number is: 10
The processed number is: 20
Error: The query has been canceled via the token supplied to
 WithCancellation.
End of the main thread.
```

**POINTS TO NOTE**

Note the following points:

1. To see similar output, you may need to adjust the sleep time to ensure the cancellation triggers while the query starts executing.
2. Notice that the main thread did not call `SomeMethod`. It was invoked by a separate task (another task). This is why the main thread did not suffer from any unhandled exception. You can also see that the `OperationCanceledException` is already handled inside `SomeMethod`. This is why you could catch this exception instantly.

**Additional Note**: The absence of this `catch` block (that handles `OperationCanceledException`) inside the `SomeMethod` could cause the following: `Unhandled exception. System.AggregateException: One or more errors occurred. (A task was canceled.)` It is because this case requires you to guard the `Wait()` call inside the main thread. Here is a sample:

```
// There is no change in the previous code
var anotherTask = Task.Run(SomeMethod, token);
//anotherTask.Wait();
try
{
```

```
    anotherTask.Wait();
}
catch (AggregateException ae)
{
 foreach (Exception e in ae.InnerExceptions)
 {
  WriteLine($"Error Type: {e.GetType().Name}, Message:
   {e.Message}");
 }
}
// There is no change in the remaining code
```

## E5.6

Here is a sample program. I kept printing intermediate outputs and task IDs for your better understanding:

```
using static System.Console;
int n = 10;
int factorial = ParallelEnumerable
    .Range(1, n)
    .Aggregate(
        // initialize subtotal/ seed value.
        1,
        // Executes this on each thread
        (partialResult, nxtNumber) =>
        {
            int temp = partialResult * nxtNumber;
            WriteLine($"The partial value: {partialResult},
             next number: {nxtNumber},  temp: {temp} id:
             [{Task.CurrentId}]");
```

```
            return temp;
        },
        // Aggregating the subtotals from all threads
        (finalResult, individualResult) =>
        {
            int temp2 = finalResult * individualResult;
            WriteLine($"The final result: {finalResult},
             individual result: {individualResult}, temp2:
             {temp2}");
            return temp2;
        },
        // Processing the final result
        total => total
    );
WriteLine($"The factorial of {n} is {factorial}");
```

Here is some sample output:

```
The partial value: 1, next number: 4, temp: 4 id: [3]
The partial value: 1, next number: 9, temp: 9 id: [5]
The partial value: 1, next number: 1, temp: 1 id: [2]
The partial value: 1, next number: 7, temp: 7 id: [4]
The partial value: 7, next number: 8, temp: 56 id: [4]
The partial value: 9, next number: 10, temp: 90 id: [5]
The partial value: 1, next number: 2, temp: 2 id: [2]
The partial value: 2, next number: 3, temp: 6 id: [2]
The partial value: 4, next number: 5, temp: 20 id: [3]
The partial value: 20, next number: 6, temp: 120 id: [3]
The final result: 6, individual result: 120, temp2: 720
The final result: 720, individual result: 56, temp2: 40320
The final result: 40320, individual result: 90, temp2: 3628800
The factorial of 10 is 3628800
```

# CHAPTER 6

# Simplifying Asynchronous Programming

Just like while preparing your breakfast you perform many activities in parallel, in the coding world while developing an application, you will want to develop applications that execute things in parallel. This is why nowadays you can listen to your favorite song while downloading another song on your mobile phone. No one wants to be blocked from doing a task while waiting until another task is complete. We all want to do multiple activities simultaneously rather than executing them one after the other. Fast-paced modern applications use asynchronous code to match real-world situations like this.

So, what is asynchronous programming? In programming terminology, we can say the following: **if a calling thread invokes a long-running task, it does not wait for its completion. Instead, it comes back to the caller to invoke another task (or continue the remaining work). Meanwhile, the invoked task will keep running. If needed, the initial caller can again go back to the previous task to continue. The process can continue until the tasks are finished.**

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5_6

In the past, asynchronous programming was considered a challenging activity, but now things are easy! C# 5.0 introduced two useful keywords to do this: `async` and `await`. This chapter discusses the programs that benefit from using them.

# Introduction to async and await

Asynchronous programming was possible in earlier versions of C#; however, you needed to perform a lot of work to make it happen. Though the Task Parallel Library (TPL) simplified things for us, there were always possibilities for improvement. The `async` and `await` keywords were introduced to fill those gaps.

To illustrate, let's consider a synchronous method that returns a number, say 100:

```
// Synchronous version
static int GetHundred()
{
    WriteLine("The method is arranging the number.");
    Task.Delay(1500);
    WriteLine("The method resumes.");
    return 100;
}
```

It is simple and easy to understand. Now let me show an asynchronous version of it using `async` and `await`. It is as follows (notice the changes in bold):

```
// Asynchronous version
static async Task<int> GetHundredAsync()
{
    WriteLine("The method is arranging the number.");
    await Task.Delay(1500);
```

```
    WriteLine("The method resumes.");
    return 100;
}
```

You can see the following changes:

- The word `Async` is added to the method name.
- The return type is now `Task<int>`.
- The method signature now includes the `async` keyword.
- There is an `await` keyword inside this method.

You may wonder about these changes. Let's discuss them one by one. I'll discuss the `await` keyword shortly. First, let's examine the `async` keyword.

**Author's Note:** I often see the terms `await` keyword, `await` expression, `await` operator, and `await` statement used interchangeably across different articles/blogs and resources. For example, if you like to see the details of the `await` keyword at https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/, you will be redirected to https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await, which uses the term `await` operator.

## Understanding async

You can use the `async` modifier to make an asynchronous method. Before we dive into the details, let's start with the naming convention.

## Naming Convention

The documentation states the following (see https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/task-asynchronous-programming-model):

> *The name of an async method, by convention, ends with an "Async" suffix.*

You understand that I have followed this naming convention for the `async` method.

## Return Types and Parameters

You may be wondering about the return type! You may think that since the return type of the asynchronous method (`GetHundred`**`Async`**) is `Task<int>`, you must create a task and return it from the method body. But instead of returning a `Task<int>`, it returned 100, which is an ordinary `int`. How was this possible? It was possible because I used the `async` keyword in the method signature.

Typically, when you mark a method with the `async` keyword, the method can have any of the following return types:

- `Task<Result>`
- `Task`
- `void`

Normally, inside an `async` method, you'll see the `await` operator as well.

**POINT TO NOTE**

Before you read further, I want you to remember the following points:

- If you use the `void` return type, the method cannot be awaited. As a result, the caller of this `void` returning method will not be able to catch any exception that the invoked method can throw.
- In addition, any type that has an accessible `GetAwaiter` method can be used as a return type as well. `ValueType<TResult>` is such an example. I'll talk about it later.
- The documentation at https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-return-types talks about a few more return types. For example, the `IAsyncEnumerable<T>` type is used for an `async` method that returns an `async` stream. This online link also talks about other types such as `IAsyncAction` and `IAsyncAction<TResult>` that are specific to Windows workloads. For now, let's make things as simple as possible and keep learning about the essentials.

I want you to remember one final point before you leave this section. If your `async` method's return type is either `void` or `Task`, the return statement is just `return` and is optional. For example, the following method can compile successfully without the `return` statement.

```
static async Task SomeTaskAsync()
{
    WriteLine("SomeTaskAsync is called.");
    await Task.Delay(1000);
    WriteLine("SomeTaskAsync is completed now.");
```

```
    // The following return statement is optional
    // return;
}
```

## Async Restrictions

While using the `async` methods, you need to remember a few more restrictions that are as follows:

- An `async` method cannot declare `in`, `ref`, or `out` parameters. However, that method can call methods that have such parameters.
- Though an `async` method can call methods with `ref` return values, it can't return a value by reference.

## Q&A Session

**Q6.1 Why does an `async` method support a limited number of types?**

Typically, you use an `async` method that awaits a long-running operation. It allows a caller to invoke the method and get the control back quickly. However, normally, when the control returns, the `async` methods are not finished and you do not know about the final result. You understand that, in this case, the incomplete method needs to resume to produce the final value. So, it makes sense that an `async` method considers `Task` or `Task<TResult>` as a return type. In addition, if you do not care about the return value, it makes sense to have a `void` return type as well.

## Understanding await

Let's discuss the `await` keyword now. Using it you can start a task in a nonblocking way and **then continue the execution when that task is completed**. Microsoft states the following (see https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await):

> *The await operator suspends evaluation of the enclosing async method until the asynchronous operation represented by its operand completes. When the asynchronous operation completes, the await operator returns the result of the operation, if any.*

Earlier, you saw the asynchronous method called `GetHundredAsync`. Now the question is how do you consume this method? The answer is as follows: `int result = await GetHundredAsync();`.

Notice that the `GetHundredAsync` method returns a `Task<int>`, but the `result` is an `int`. How does the compiler allow this? The `await` keyword has done this magic by unwrapping from `Task<int>` to `int`.

The next question is if `GetHundredAsync` is a time-consuming method, how can you write better code? See the following sample, which comes from the upcoming demonstration:

```
Task<int> getValue = GetHundredAsync();
WriteLine("The main thread continues.");
// Consuming the return value which is an int
int result = await getValue;
WriteLine($"The invoked task returns: {result}");
```

Let's understand the flow now. First, the calling thread invokes `GetHundredAsync`. So, the control goes there and starts executing. After going there, when it sees the line `await Task.Delay(1500);`, the enclosing `async` method (i.e., `GetHundredAsync`) is suspended (control resumes here when the delay is over). Meanwhile, the control comes back to the calling thread. So, the calling thread becomes free.

As a result, the calling thread starts executing again until it needs the return value of `GetHundredAsync`. This is why it can now process the line `WriteLine("The main thread continues.");`.

Next, the calling thread sees the following line:

```
int result = await getValue;
```

Since this line contains the `await` operator, this time the calling thread needs to wait for the completion of the `getValue` task. Once the task is completed, the calling thread can resume. This means you do not need to call the `Start` method on the task explicitly (yes, it is still hot!).

**POINT TO REMEMBER**

If you mark a method as `async`, you can apply the `await` keyword. Then the compiler will do all the heavy work for you to ensure that the method gets split into smaller pieces. In other words, the `await` keyword helps you mark a point in the method where that method can be suspended or split up.

## Demonstration 1

It's time for a demonstration to cross-verify our understanding. Here is the complete program:

```
using static System.Console;

WriteLine("The main thread starts.");
```

```
Task<int> getValue = GetHundredAsync();
WriteLine("The main thread continues.");
// Consuming the return value which is an int
int result = await getValue;
WriteLine($"The invoked task returns: {result}");
WriteLine("The main thread ends.");

// Asynchronous version
static async Task<int> GetHundredAsync()
{
    WriteLine("The method is arranging the number.");
    await Task.Delay(1500);
    WriteLine("The method resumes.");
    return 100;
}
```

## Output

Here is some sample output:

```
The main thread starts.
The method is arranging the number.
The main thread continues.
The method resumes.
The invoked task returns: 100
The main thread ends.
```

## Analysis

Let me remind you that in the case of asynchronous programming, we do not want the calling thread to be blocked. If the calling thread is not blocked, how can the invoked method (which is called by the calling thread) continue? You understand that we need another thread for the invoked method to continue.

To investigate this, let's put some breakpoints in this program and analyze them. First, while debugging, launch the thread window by selecting the **Debug** menu, selecting **Windows**, and then selecting **Threads**, as shown in Figure 6-1.

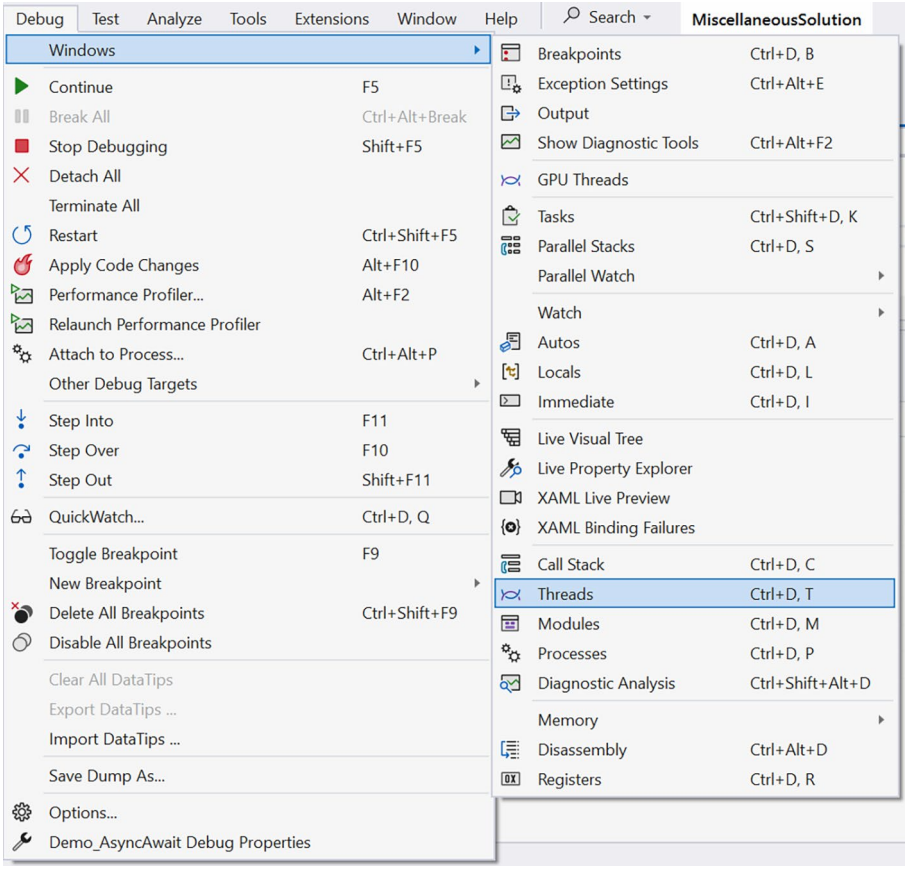



***Figure 6-1.*** *Launching the Threads window while debugging*

Notice that while the control (aka instruction pointer) hits the line `WriteLine("The invoked task returns:{result}");`, the following snapshot shows that both the main thread and the worker thread are running (see Figure 6-2).

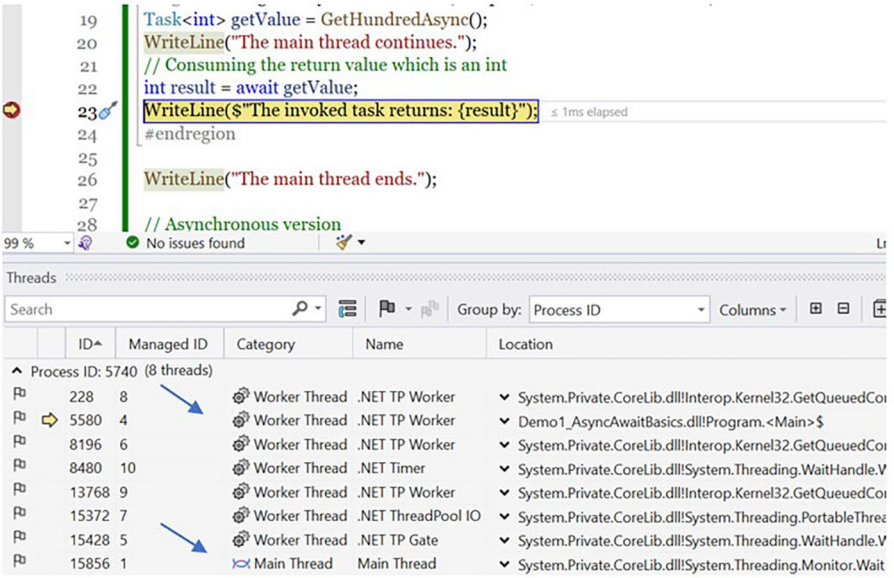

19 Task<int> getValue = GetHundredAsync();
20 WriteLine("The main thread continues.");
21 // Consuming the return value which is an int
22 int result = await getValue;
23 WriteLine($"The invoked task returns: {result}");
24 #endregion
25
26 WriteLine("The main thread ends.");
27
28 // Asynchronous version
99 %
No issues found
Threads
Search
Group by: Process ID
Columns
ID
Managed ID
Category
Name
Location
Process ID: 5740 (8 threads)
228 8 Worker Thread .NET TP Worker System.Private.CoreLib.dll!Interop.Kernel32.GetQueuedCo
5580 4 Worker Thread .NET TP Worker Demo1_AsyncAwaitBasics.dll!Program.<Main>$
8196 6 Worker Thread .NET TP Worker System.Private.CoreLib.dll!Interop.Kernel32.GetQueuedCo
8480 10 Worker Thread .NET Timer System.Private.CoreLib.dll!System.Threading.WaitHandle.V
13768 9 Worker Thread .NET TP Worker System.Private.CoreLib.dll!Interop.Kernel32.GetQueuedCo
15372 7 Worker Thread .NET ThreadPool IO System.Private.CoreLib.dll!System.Threading.PortableThrea
15428 5 Worker Thread .NET TP Gate System.Private.CoreLib.dll!System.Threading.WaitHandle.V
15856 1 Main Thread Main Thread System.Private.CoreLib.dll!System.Threading.Monitor.Wait

***Figure 6-2.*** *The main thread is running along with the worker thread*

Figure 6-2 shows that multiple threads start running while the program executes.

## Q&A Session

**Q6.2 What do you mean by a "hot" task?**

A task is *hot* if you do not need to call the `Start` method on the `Task` object. In technical terms, these are initiated and their task status is an enumeration value other than `TaskStatus.Created`. So, these are active tasks.

When you create `Task` instances using the public `Task` constructors, those are cold tasks because they are scheduled only when you call the `Start` method on those instances.

**Q6.3 I can see that multiple threads are involved in the previous program. Does this mean that asynchronous programming inherently uses multithreading?**

Yes.

**Q6.4 When should I prefer the `void` return type over the `Task` return type (and vice versa)?**

Note that a returned task represents ongoing work. A task encapsulates the required information about the state of the process. It is often useful to know the final outcome because any process can raise an exception. To illustrate, consider the following program:

```
using static System.Console;

WriteLine("The main thread starts executing.");
Task getTask=SomeTaskAsync();
WriteLine("The main thread resumes.");
await getTask;

static async Task SomeTaskAsync()
{
    WriteLine("SomeTaskAsync is called.");
    await Task.Delay(1000);
    WriteLine("SomeTaskAsync is completed now.");
    // The following return statement is optional
    // return;
}
```

Here is some sample output from this program:

```
The main thread starts executing.
SomeTaskAsync is called.
The main thread resumes.
SomeTaskAsync is completed now.
```

You can see that the program lets you know when the invoked task (`SomeTaskAsync`) was completed. This is because you awaited the returned task (see the line `await getTask;`) to know the final result.

---

**Note** You can download the `Discussing_AsyncReturnTypes` project to exercise this program.

---

However, if you opt for the `void` return type (instead of `Task`), the caller may not know about the final result of the invoked task. So, the `void` return type is useful when you do not care about a result or you do not show any interest in knowing whether the method executes successfully. This is why its usage is rare. However, the `void` return type can be useful for the event handlers that can be used as a "fire-and-forget" operation such as initiating an `async` program.

**POINT TO NOTE**

Stephen Cleary's blog is a nice resource for learning (see https://blog.stephencleary.com/2013/01/async-oop-0-introduction.html). Though it was written a long time ago, I still believe that reading those materials can help you a lot. You may note that he likes to use the term "fire-and-forget" sensibly. In reply to a question on stack overflow (see https://stackoverflow.com/questions/17659603/async-void-asp-net-and-count-of-outstanding-operations/17660475#17660475), he said the following: "I personally never use this phrase for async void methods. For one thing, the error handling semantics most certainly do not fit in with the phrase 'fire and forget'; I half-jokingly refer to async void methods as 'fire and crash'. A true async 'fire and forget' method would be an async Task method where you ignore the returned Task rather than waiting for it."

## Await Restrictions

While using the `await` keyword in your code, you need to remember the following restrictions:

**Restriction 1: You cannot use `await` without `async` inside a method, lambda expression, or anonymous method**. For example, if you write something like the following:

```
class Sample
{
    public void SomeMethod()
    {
        // Some code, if any
        await Task.Delay(1000); //Error CS4033
        // Some other code, if any
    }
}
```

you'll see the following error:

```
Error CS4033 The 'await' operator can only be used within an
async method. Consider marking this method with the 'async'
modifier and changing its return type to 'Task'.
```

**Restriction 2: You cannot use `await` in an unsafe context**. For example, if you write something like the following:

```
unsafe
{
    // Some code, if any
    await Task.Delay(1000); // Error CS4004

    // Some code
    // Task.Delay(1000); // OK
}
```

you'll see the following error:

```
Error CS4004 Cannot await in an unsafe context
```

**Restriction 3: You cannot use await inside a lock.** For example, if you write something like the following:

```
class Foo
{
    private readonly object _padLock = new();
    public async void SomeMethod()
    {
        lock (_padLock)
        {
            // Do the synchronous operations, if any
            // Trying to do an asynchronous operation now
            await Task.Delay(1000); // Error CS1996
        }
    }
}
```

you'll see the following error:

```
Error CS1996 Cannot await in the body of a lock statement
```

**Finally, I want you to note the following code block:**

```
try
{
    // do something
    throw new Exception("Forceful exception.");
}
catch (Exception e)
{
    await Task.Delay(1000);
```

```
    // Do something else
}
finally
{
    await Task.Delay(1000);
    // Do something else
}
```

From C# 6.0 onward this code will compile. However, C# 5.0 will complain about it.

---

**Note** You can download the `Await_Restrictions` project to exercise these code segments.

---

## Q&A Session

**Q6.5 I understand that I cannot use the `await` operator inside a lock. However, I'll like to know whether there is any workaround.**

The very first thing you should understand is how the asynchronous programming model works with the `async` and `await` keywords. You may try to design your application in a way that can avoid this need. Otherwise, you can try the following verbose code where I used the lock statement multiple times and processed the asynchronous code separately (I kept the comments for your reference):

```
public async void SomeMethod()
{
     lock (_padLock)
     {
         // Do the synchronous operations
         // and prepare for the asynchronous operation
```

```
    }

    // Doing the asynchronous operation now
    await Task.Delay(1000);
    // Lock again, if required
    lock (_padLock)
    {
        // Do the remaining synchronous operations
    }
}
```

## State Machines Are Behind the Scenes

Consider the following program where you see the use of three `await` keywords. One of them is at the beginning of the file, and the other two are placed inside the `ExecuteLongTaskAsync()` method.

```
using System;
using System.Threading.Tasks;
using static System.Console;

WriteLine("Main thread starts.");

var getValue = ExecuteLongTaskAsync();
WriteLine("Main thread continues.");
int result = await getValue;
WriteLine($"The long task returns: {result}");
WriteLine("Main thread ends.");

static async Task<int> ExecuteLongTaskAsync()
{
    WriteLine("Long running method starts.");
    Random random = new();
```

```
    await Task.Delay(500);
    WriteLine("Long running method resumes.");
    await Task.Delay(700);
    WriteLine("Long running method resumes again.");
    return random.Next(1, 100);
}
```

Let's investigate the IL code. To show you the IL code, I used **SharpLab** (https://sharplab.io/) here. First, I want you to notice the bold portions in the following segment (do not worry; I'll show you better readable code shortly):

```
// Previous lines not shown

.class nested private auto ansi sealed beforefieldinit
'<<Main>$>d__0'
        extends [System.Runtime]System.Object
        implements [System.Runtime]System.Runtime.CompilerServices.
         IAsyncStateMachine
    {
        .custom instance void [System.Runtime]System.Runtime.
         CompilerServices.CompilerGeneratedAttribute::.ctor() = (
            01 00 00 00
        )
        // Fields
        .field public int32 '<>1__state'
        .field public valuetype [System.Runtime]System.Runtime.
         CompilerServices.AsyncTaskMethodBuilder '<>t__builder'
        .field public string[] args
        .field private class [System.Runtime]System.Threading.
         Tasks.Task`1<int32> '<getValue>5__1'
        .field private int32 '<result>5__2'
```

```
.field private int32 '<>s__3'
.field private valuetype [System.Runtime]System.
 Runtime.CompilerServices.TaskAwaiter`1<int32> '<>u__1'

// Methods
.method public hidebysig specialname rtspecialname
    instance void .ctor () cil managed
{
    // Method begins at RVA 0x20e9
    // Code size 8 (0x8)
    .maxstack 8

    IL_0000: ldarg.0
    IL_0001: call instance void [System.Runtime]
     System.Object::.ctor()
    IL_0006: nop
    IL_0007: ret
} // end of method '<<Main>$>d__0'::.ctor

// Remaining code skipped
```

In this code, you notice a sealed class, named `<<Main>$>d__0` (but not the `Main` method). You can also see that this class extends `Object` and implements `IAsyncStateMachine`. What does this mean? It shows that behind the scenes, there is a state machine. **So, the first thing to understand is that the compiler has generated a temporary object for you**. On further investigation, you'll understand the following things:

- These fields exist for the variables inside the function.
- These states will be preserved on the state machine.

Let's learn about the `IAsyncStateMachine` interface. Visual Studio shows the following details:

```
namespace System.Runtime.CompilerServices
```

```
{
    //
    // Summary:
    // Represents state machines that are generated for
    // asynchronous methods. This type is intended for compiler
    // use only.
    public interface IAsyncStateMachine
    {
        //
        // Summary:
        // Moves the state machine to its next state.
           void MoveNext();
        //
        // Summary:
        // Configures the state machine with a heap-allocated
            replica.
        //
        // Parameters:
        // stateMachine:
        // The heap-allocated replica.
           void SetStateMachine(IAsyncStateMachine stateMachine);
    }
}
```

The associated comments are self-explanatory. For now, our focus will be on the `MoveNext` method in the decompiled code.

Let's investigate the other details now. Since the IL code is not readable enough, for our further investigation, let's choose C# (instead of IL) under Decompile in SharpLab as shown in Figure 6-3.

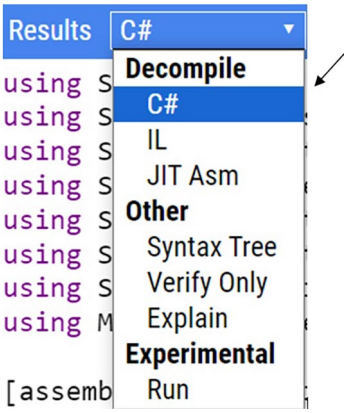



***Figure 6-3.*** *Opting for the C# option instead of IL under the Decompile tab in SharpLab*

This time you can see a better readable code. Let's first investigate the entry points of the code where you saw one `await` expression. Figure 6-4 shows a partial snapshot of this. You can see that it contains a `TaskAwaiter<int>` instance called `awaiter` in the `MoveNext` method as shown in Figure 6-4.

```
[StructLayout(LayoutKind.Auto)]
[CompilerGenerated]
private struct <<Main>$>d__0 : IAsyncStateMachine
{
    public int <>1__state;

    public AsyncTaskMethodBuilder <>t__builder;

    private TaskAwaiter<int> <>u__1;

    private void MoveNext()
    {
        int num = <>1__state;
        try
        {
            TaskAwaiter<int> awaiter;
            if (num != 0)
            {
                Console.WriteLine("Main thread starts.");
                Task<int> task = <<Main>$>g__ExecuteLongTaskAsy
                Console.WriteLine("Main thread continues.");
                awaiter = task.GetAwaiter();
                if (!awaiter.IsCompleted)
                {
                    num = (<>1__state = 0);
                    <>u__1 = awaiter;
```

***Figure 6-4.*** *One TaskAwaiter<int> instance is present inside the MoveNext method*

Now focus on the `ExecuteLongTaskAsync()` method and notice the arrow tips shown in Figure 6-5.

***Figure 6-5.*** *Two TaskAwaiter<int> instances are present inside the MoveNext method*

You can see that this time there are two `TaskAwaiter` instances (`awaiter` and `awaiter2`) inside the `MoveNext()` method. These two instances reflect that there are two `await` keywords inside the `ExecuteLongTaskAsync` method.

**POINT TO NOTE**

Now you understand that if a method contains one `await` keyword, the method is divided into two parts. If there are multiple `await` keywords inside the method, the method is divided into multiple parts accordingly.

Next, refer to the arrows in Figure 6-6.

```
{
    if (num == 1)
    {
        awaiter = <>u__1;
        <>u__1 = default(TaskAwaiter);
        num = (<>1__state = -1);
        goto IL_00fb;
    }
    Console.WriteLine("Long running method starts.");
    <random>5__1 = new Random();
    awaiter2 = Task.Delay(500).GetAwaiter();
    if (!awaiter2.IsCompleted)
    {
        num = (<>1__state = 0);
        <>u__1 = awaiter2;
        <<<Main>$>g__ExecuteLongTaskAsync|0_0>d stateMachine = this;
        <>t__builder.AwaitUnsafeOnCompleted(ref awaiter2, ref stateMachine);
        return;
    }
}
else
{
    awaiter2 = <>u__1;
    <>u__1 = default(TaskAwaiter);
    num = (<>1__state = -1);
}
awaiter2.GetResult();
Console.WriteLine("Long running method resumes.");
awaiter = Task.Delay(700).GetAwaiter();
if (!awaiter.IsCompleted)
{
    num = (<>1__state = 1);
    <>u__1 = awaiter;
    <<<Main>$>g__ExecuteLongTaskAsync|0_0>d stateMachine = this;
    <>t__builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
    return;
}
goto IL_00fb;
IL_00fb:
awaiter.GetResult();
Console.WriteLine("Long running method resumes again.");
result = <random>5__1.Next(1, 100);
}
catch (Exception exception)
```

***Figure 6-6.*** *The `ExecuteLongTaskAsync()` method is sliced through awaiter2 and awaiter*

Figure 6-6 reveals the following observations:

- Notice the code segment where you see the `if` block: `if (!awaiter2.IsCompleted){..}`. It shows that if `awaiter2` is not completed, there is a certain logic to ensure that once the thread is back, it can resume the execution from the intended point. If you scroll down, you can also notice that a similar logic is implemented for `awaiter` as well. Anyway, you understand that these are things by which states are preserved, and they help a thread to resume execution from an intended point.
- You can see that the line `Long running method resumes.` appears after the line `awaiter2.GetResult()`. You can also see that the line `Long running method resumes again.` comes after the line `awaiter.GetResult()`. This code format indicates that once a particular `awaiter` is completed, you can get the result and process the next part.

**POINT TO NOTE**

The previous bullet point indicates that the `await` keyword slices the method into parts.

## Q&A Session

**Q6.6 How can I distinguish between asynchronous and parallel programming?**

Let's first consider some real-world examples. Suppose you set up an office that delivers different products to customers. In your office, there is a customer care executive. When a client dials the customer support

number, this executive officer picks up the call, listens to the issue, and tries to solve the issue. Now think of a situation when multiple customers try to get help almost at the same time. Obviously, one of them can connect to the executive, but others need to wait. How do you interpret this situation? Surely, it's not a pleasant experience for your customers. So, you decide to hire another customer care executive to serve your customers fast. For example, if there are two executives, those two officers can serve two customers at the same time, aka in parallel. This is equivalent to parallel programming with a dual-core machine.

Now imagine a slightly different situation where the customer care executive listens to the problem, takes a note, and provides a ticket number to the customer. He also ensures the customer that the problem will be solved (or the ticket will be resolved) as soon as possible and that the company will inform the customer about the resolution via an email or a phone call. You can see that, in this case, though the problem is not solved immediately, the customer care executive can serve more customers in a specified time frame. This is equivalent to asynchronous programming. The benefit is obvious: the customers do not need to stay on the call until the problem is solved. As a result, the customer can do other work while the company executive tries to find a solution to the problem.

Another real-life situation in this category is when you order something from a grocery store and then start checking your email. You know the delivery person will deliver the order on time and you do not need to stop working while waiting for it.

So, if you combine these two models, the response time can be very fast. For example, if multiple support executives listen to multiple customers, provide some tickets, and assure them about the resolutions, the customers will have a better experience. We try to do this in programming. In fact, the terms *asynchronous programming* and *parallel programming* are related to each other. The Visual Studio Magazine summarized this as follows (see https://visualstudiomagazine.com/articles/2011/03/24/wccsp_asynchronous-programming.aspx):

> *Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress.*

**Q6.7 Seeing your explanation, I get the feeling that in the previous chapters we also exercised asynchronous programming. Is this correct?**

True. In fact, when you create a new thread from a calling thread and run both threads, surely you are doing things asynchronously. But the focus of this chapter is on the `async` and `await` keywords, particularly how to simplify asynchronous programming using these keywords.

**Author's Note:** This `async-await` programming model in C# can be called other things in other programming languages. For example, C++ developers often use the term *coroutines* in a similar context. The wiki page at `https://en.wikipedia.org/wiki/Coroutine` states the following:

> *C# 2.0 added semi-coroutine (generator) functionality through the iterator pattern and yield keyword. C# 5.0 includes await syntax support.*

# Task.Run vs. Task.Factory.StartNew

This section compares the `Task.Run` and `Task.Factory.StartNew` methods using some small and valid code segments. In these code segments, my focus will be on the return types. To make things fast, instead of using the `var` type and then using the `GetType` method to determine the type, I'll explicitly show you the type in these code segments. Let's begin.

## Task.Factory.StartNew Treats Ordinary Lambdas and Asynchronous Lambdas Differently

Notice that in Code Segment 1, `task1` is of type `Task<int>`. You can see that the return type `int` is wrapped with an additional layer.

**Code Segment 1:**

```
// Notice that task1 is a Task<int>
Task<int> task1 = Task.Factory.StartNew(
    () =>
    {
        return 100;
    }
);
```

**Code Segment 2:**

Now I make the lambda `async`. As a result, the return type becomes `Task<Task<int>>`. You can see that the return type `int` is now wrapped with one more additional layer.

```
// Notice that task2 is a Task<Task<int>> [ for applying async]
Task<Task<int>> task2 = Task.Factory.StartNew(
    async() =>
    {
        return 100;
    }
);
```

These code segments show that `Task.Factory.StartNew` treats ordinary lambdas and asynchronous lambdas differently.

## Task.Run Treats Ordinary Lambdas and Asynchronous Lambdas Uniformly

Now start analyzing similar code with the `Task.Run` method. For example, see the following code segments (Code Segments 3 and 4) with supportive comments. In both cases, you see the return type as `Task<int>`.

**Code Segment 3:**

```
// Notice that task3 is a Task<int>.

Task<int> task3 = Task.Run(
    () =>
    {
        return 100;
    }
);
```

**Code Segment 4:**

```
// Notice that task4 is also a Task<int>.

Task<int> task4 = Task.Run(
    async () =>
    {
        return 100;
    }
);
```

You can see that when you pass an `async` lambda inside the `Run` method, there is no change in the return type.

**POINT TO NOTE**

Code Segments 3 and 4 show that `Task.Run` treats ordinary lambdas and asynchronous lambdas uniformly.

## Unwrapping Using the Unwrap Method

While using the `StartNew` method, you can apply the `Unwrap` method as follows:

**Code Segment 5:**

```
Task<int> task5 = Task.Factory.StartNew(
    async () =>
    {
        return 100;
    }
).Unwrap();
```

See that you have unwrapped from `Task<Task<int>>` to `Task<int>`.

## Unwrapping Using await

You can use `await` as an unwrapper. For example, see the following code:

**Code Segment 6:**

```
// await is unwrapping the layer
int task6 = await Task.Run(
    () =>
    {
        return 100;
    }
);
```

You can see that task6 is now an int, but not a Task<int>.

**Code Segment 7:**

As you have already seen, the Run method treats ordinary lambdas and asynchronous lambdas uniformly. Applying await in the following code helps you get the return type int again.

```
// await is unwrapping the layer
int task7 = await Task.Run(
    async() =>
    {
        await Task.Delay(100);
        return 100;
    }
);
```

---

**Note** You can test these code segments by downloading the project UnderstandingTaskDotRun from the Apress website.

---

# Asynchronous Construction

While using the async and await keywords, you need to follow certain restrictions. Acknowledging them will help you understand this section easily. Let's start.

For our upcoming discussions, I'll use a class called Repository that is defined as follows:

```
class Repository
{
    public static async Task<int> GetDataAsync()
    {
```

```
        await Task.Delay(1000);
        return new Random().Next(1, 10);
    }
}
```

Given this code, **the C# compiler does not complain if you invoke a task inside a constructor as follows:**

```
class Sample
{
    private int _flag;
    public Sample()
    {
        Task.Run(
           async () =>
           {
               _flag = await Repository.GetDataAsync();
               WriteLine($"The flag is set to {_flag} now");
           }
           );
    }
}
```

**Interestingly, the C# compiler also allows you to call an asynchronous method from the constructor**. This is why the following code will compile as well (Later, you'll see that this is not recommended):

```
class Sample
{
    public int Flag { get; set; }

    private async void InitializeAsync()
    {
```

```
        Flag = await Repository.GetDataAsync();
        WriteLine($"The flag is set to {Flag} now");
    }
    public Sample()
    {
        InitializeAsync();
    }
}
```

**However, C# does not allow you to mark a constructor with the `async` keyword. This means you cannot use the `await` keyword in the constructor too**. For your reference, consider the following code that will not compile (you'll see multiple compile-time errors for this code segment):

```
class Sample
{
    public int Flag { get; set; }

    // Error
    public async Sample()
    {
        Flag = await Repository.GetDataAsync();
        WriteLine($"The flag is set to {Flag} now");
    }

}
```

Then how can you perform an asynchronous initialization? Let's investigate some possible approaches and the associated pros and cons.

**POINT TO REMEMBER**

Using the `await` operator, you can wait until the `async` operation is completed. When an `async` method returns a `Task` or `Task<T>`, you can handle the exceptions easily. Since you cannot use the `async` and `await` operators inside the constructor, asynchronous initialization is a challenging activity. This section discusses the possible approaches that can handle the situation.

# Approach 1

Since you cannot construct everything asynchronously, you can attempt to segregate the asynchronous initialization part from its counterpart. For example, you call a constructor to perform the usual initialization (for the synchronous part), and then you invoke a method that can perform the asynchronous initialization (for the asynchronous part).

## Demonstration 2

Here is a sample demonstration:

**POINT TO NOTE**

I made the `InitializeAsync` method's return type `Task<Sample>`. It is because you cannot `await` on this method if its return type is `void`.

```
using static System.Console;

WriteLine("Examining an async initialization.");
```

```
Sample sample = new();
// Performing async initialization and unwrapping Task<Sample>
// to Sample
sample = await sample.InitializeAsync();
sample.GetFlagValue();

class Sample
{
    private int _flag;

    public Sample()
    {
        // Performing sequential construction here
    }

    public async Task<Sample> InitializeAsync()
    {
        _flag = await Repository.GetDataAsync();
        WriteLine($"The flag is set to {_flag} now.");
        // Some other code, if any
        return this;
    }

    public void GetFlagValue()
    {
        WriteLine($"The current flag value is {_flag}.");
    }
}

// The Repository class is not shown again to avoid repetition.
```

## Output

Here is some sample output from my computer:

```
Examining an async initialization.
The flag is set to 1 now.
The current flag value is 1.
```

## Analysis

You can see that the usage of the `Sample` class is troublesome. This is because you need to use a traditional constructor along with the intended method for the complete initialization of a class. If you forget to call any of them, you will see an unintended result. This is misleading in the sense that a constructor did not perform the complete initialization. In addition, you cannot invoke the `async` initialization part before getting a `Sample` instance that performs the sequential initialization. So, **this is not a recommended approach.**

# Approach 2

You already learned that the C# compiler allows you to call an asynchronous method from the constructor. Let's use this concept in the following program and analyze it.

## Demonstration 3

Here is a sample program:

```
using static System.Console;

WriteLine("Examining an async initialization.");

Sample sample = new();
sample.GetFlagValue();
```

```
ReadKey();

class Sample
{
    private int _flag;
    public Sample()
    {
        InitializeAsync();
    }
    private async void InitializeAsync()
    {
        _flag = await Repository.GetDataAsync();
        WriteLine($"The flag is set to {_flag} now.");
    }

    public void GetFlagValue()
    {
        WriteLine($"The current flag value is {_flag}.");
    }
}
// The Repository class is not shown again to avoid repetition.
```

## Output

Here is some sample output from my computer (I did not press any key until I saw the last line of output):

```
Examining an async initialization.
The current flag value is 0.
The flag is set to 8 now.
```

## Analysis

You can see that when the `GetFlagValue` method was called, the updated flag value (`_flag`) was not set. This is because when the constructor completed, the instance was still being asynchronously initialized, and you did not know when the asynchronous initialization would complete.

By introducing a delay before calling the `GetFlagValue` method, you can increase the possibility of getting the intended output. To illustrate, when I introduce a two-second delay before I invoke the `GetFlagValue` method as follows:

```
using static System.Console;

WriteLine("Examining async initialization");

Sample sample = new();
Thread.Sleep(2000);
sample.GetFlagValue();

//There is no other change in the remaining code
```

I see the intended output on my computer. Here is a sample for your reference:

```
Examining an async initialization.
The flag is set to 7 now.
The current flag value is 7.
```

However, you know the purpose of a constructor. When a call returns from the constructor, you expect to see the fully initialized object. It should not be the case that you get the fully initialized object at some undefined point of time in the future. At the same time, you also expect that the initialization process will not encounter an exception. Otherwise, you must handle it properly.

To illustrate, suppose you surround the object construction code inside a try-catch block and exercise the code as follows:

```
WriteLine("Examining async initialization");

try
{
    Sample sample = new();
    Thread.Sleep(2000);
    sample.GetFlagValue();
}
catch (Exception e)
{
    WriteLine(e.Message);
}
ReadKey();

class Sample
{
    private int _flag;
    public Sample()
    {
        InitializeAsync();
    }
    private async void InitializeAsync()
    {
        _flag = await Repository.GetDataAsync();
        WriteLine($"The flag is set to {_flag} now.");
        // The calling thread cannot catch the following
        // exception which can be raised due to some logic
        // throw new InvalidOperationException("Invalid ops!");
    }

// There is no change in the remaining code
```

This `try-catch` block is not sufficient to handle the exception that can be raised by the `InitializeAsync` method (notice that the asynchronous method has the `void` return type). You can see that handling exceptions inside the `InitializeAsync` method is another concern. So, **this is not a recommended approach either.**

Let's see another approach now.

# Approach 3 (Using a Static Method aka Factory)

To address the limitations of the previous approaches, you can employ the factory pattern. Let's analyze it further.

## Demonstration 4

This program uses a static method to perform the complete initialization. Here is a sample program:

```
using static System.Console;

WriteLine("Examining an async initialization.");

// Sample sample = new(); // Error now

// Performing async initialization and unwrapping Task<Sample>
// to Sample
Sample sample = await Sample.InitializeAsync();
sample.GetFlagValue();

class Sample
{
    private int _flag;
    private Sample()
```

```
    {
        // Do something here, if required
    }

    // Doing the initialization work here
    public static async Task<Sample> InitializeAsync()
    {
        Sample sample = new()
        {
            _flag = await Repository.GetDataAsync()
        };

        WriteLine($"The flag is set to {sample._flag} now.");
        return sample;
    }

    public void GetFlagValue()
    {
        WriteLine($"The current flag value is {_flag}.");
    }
}

// The Repository class is not shown again to avoid repetition.
```

## Output

Here is some sample output:

```
Examining an async initialization.
The flag is set to 2 now.
The current flag value is 2.
```

## Analysis

You can see that this solves the problems of the previous approaches. Notice that in this program, the `GetFlagValue` method always displays the updated value of the `flag` variable (`_flag`). In addition, if you surround the following code using a `try-catch` block inside the main thread as follows:

```
try
{
    // Performing async initialization and unwrapping
    // Task<Sample> to Sample
    Sample sample = await Sample.InitializeAsync();
    sample.GetFlagValue();
}
catch (Exception e)
{
    WriteLine(e.Message);
}
```

then you can catch any exception that can be raised by the `InitializeAsync` method. This is a common approach for an asynchronous construction.

## Q&A Session

**Q6.8 I can see that you have made the constructor private. Was this intentional?**

Yes. In this case, I'd also like to prohibit the direct usage of the constructor so that readers know there is only one way to initialize. Interestingly, if you are familiar with the singleton design pattern, you are already familiar with this approach.

# Approach 4 (Lazy Initialization)

If an object creation is a time-consuming or costly operation, you may delay the instantiation process. The idea is that unless you need the object, you should avoid the construction process. This tactic is helpful when you deal with shared resources that are typically large.

## Understanding Lazy<T>

Surely, you can employ a custom logic to implement the concept. However, the good news is that the `Lazy<T>` class can help in this context. Let's get a quick overview of it.

This time the `Sample` class has the following look:

```
class Sample
{
    public int Flag { get; init; }
    public Sample(int  flag)
    {
        Flag = flag;
    }
    // It also can have some big data such as:
    // public int[] Data = new int[5000];

}
```

You can always make an object of `Foo` using this approach: `var sample2 = new Foo(10);`.

When you'd like to construct an object of `Foo` lazily, you will see several constructors. At the time of this writing, I can see what's shown in Figure 6-7.

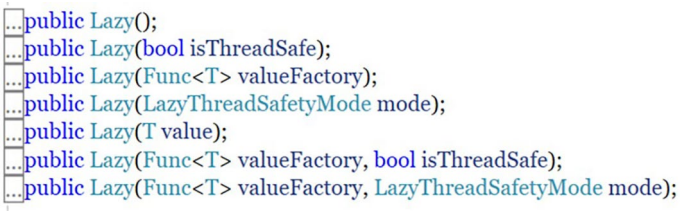



***Figure 6-7.*** *Available constructors for lazy initializations*

To instantiate a `Sample` object lazily, let's use the third constructor as follows:

```
var sample1 = new Lazy<Sample>(
    () => { return new Sample(5); }
    );
```

**In this case, the large object (i.e., `sample1`) is not constructed until you access the `Flag` property. By using `Value` (gets the lazily initialized value of the current `Lazy<T>` instance) inside a lazily initialized object, you can access the class properties (in this example, it is `Flag`).**

## Demonstration 5

Let's debug the following program to verify this:

```
using static System.Console;

var sample1 = new Lazy<Sample>(
    () => { return new Sample(5); }
    );

var sample2 = new Sample(10);
ReadKey();
WriteLine(sample1.Value.Flag);
ReadKey();

class Sample
{
```

```
    public int Flag { get; init; }
    public Sample(int  flag)
    {
        Flag = flag;
    }
    // It also can have some big data such as:
    // public int[] Data = new int[5000];
}
```

## Analysis

Figure 6-8 shows when the control hits the breakpoint at line 8.

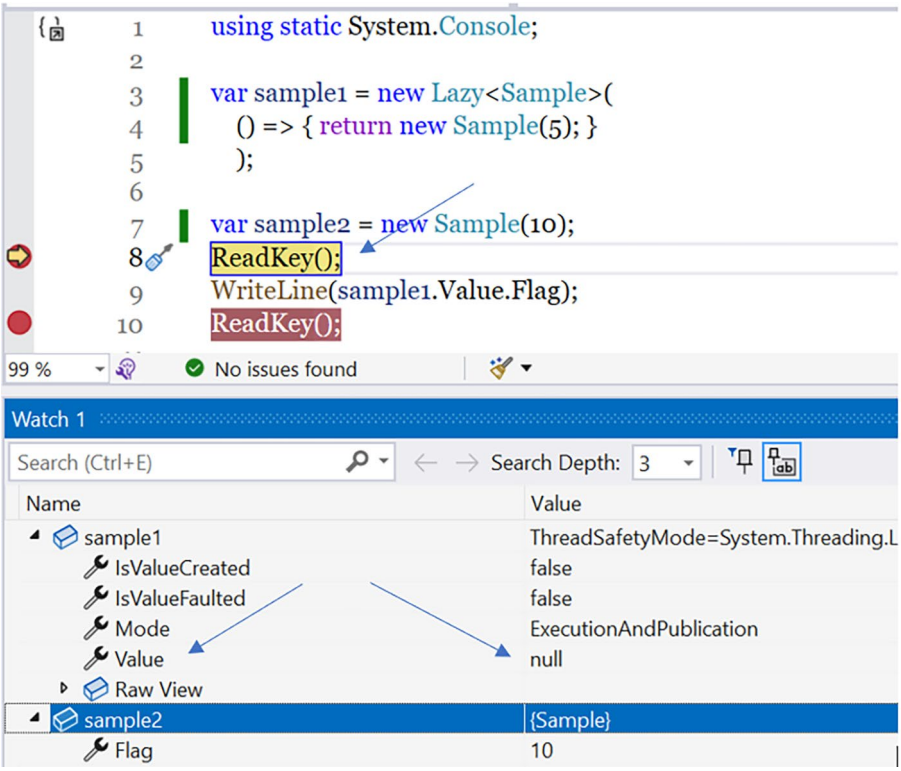



***Figure 6-8.*** *The sample2 instance is ready, but the sample1 instance is not ready yet*

You can see that the `sample2` instance was created, but the `sample1` instance is not ready yet. In addition, notice that the `Value` property of the `sample1` instance is `null` at this stage.

Let's continue the execution. (Since I used `ReadKey` to pause the execution, you need to press F5 and any key to proceed further.) Now control hits the breakpoint at line 10. This time notice the Watch window as show in Figure 6-9.

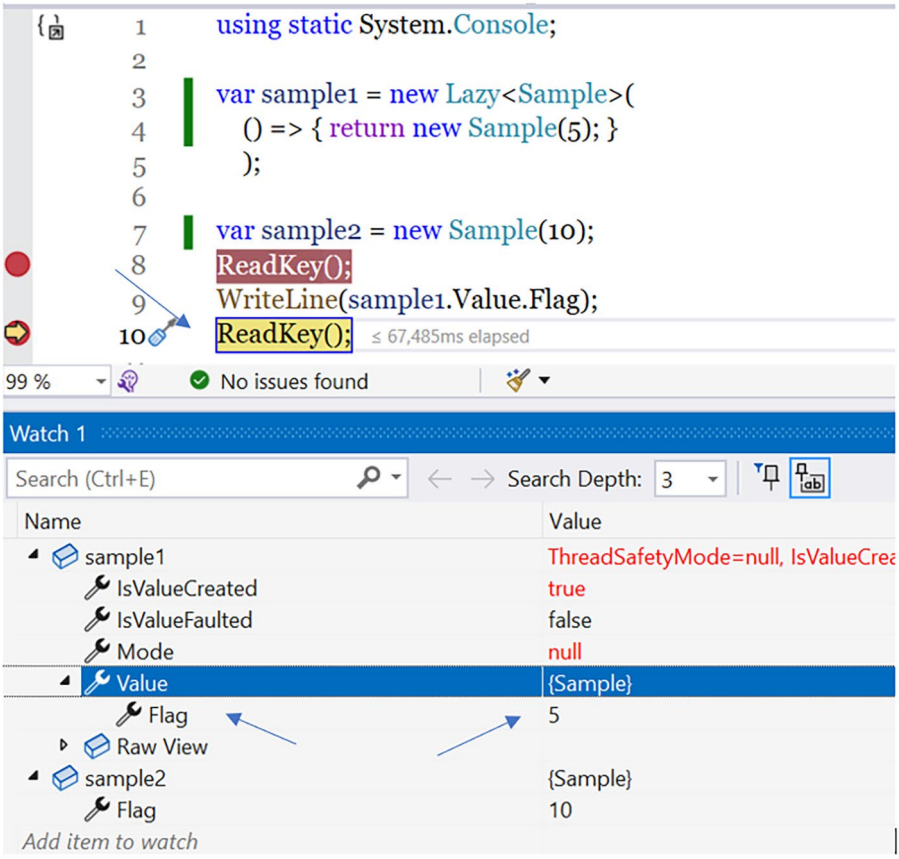



***Figure 6-9.*** *The construction process of the sample1 instance is completed now*

You can see that once you access the `Value` property, the `sample1` instance is constructed. This is why the `Value` property of the `sample1` instance is not null anymore.

## Using Lazy<Task<T>> for Async Construction

You saw `Lazy<Sample>` in the previous demonstration. However, note that `Lazy<T>` **does not support asynchronous initialization**. This causes obvious problems when the initialization involves long-running tasks or I/O operations. For example, while downloading a large file from a remote server, blocking the main thread will cause a poor user experience for sure.

**Why doesn't `Lazy<T>` support asynchronous initialization?** At https://devblogs.microsoft.com/pfxteam/asynclazyt/, Stephen Toub, a partner software engineer from Microsoft, answers this question as follows:

> *Lazy<T> is all about caching a value and synchronizing multiple threads attempting to get at that cached value, whereas we have another type in the .NET Framework focused on representing an asynchronous operation and making its result available in the future: Task<T>. Rather than building asynchronous semantics into Lazy<T>, you can instead combine the power of Lazy<T> and Task<T> to get the best of both types!*

Let's exercise an asynchronous initialization using `Lazy<Task<Sample>>` now. In the upcoming program, you'll see the following code block inside the `Sample` class:

```
private static Lazy<Task<Sample>> InitializeAsync =
   new(
          async () =>
          {
              int data = await Repository.GetDataAsync();
              return new Sample(data);
          }
      );
```

**POINT TO NOTE**

Notice that I return a `Sample` object that is wrapped as `Task<Sample>`. Why? This is because there is an `async` lambda, and it does this wrapping, which you already learned about. In other words, if the lambda is not `async`, then the program will intend to work with `Lazy<Sample>`, but not with `Lazy<Task<Sample>>`.

Using this code block is easy. You'll see the following `public` method (inside the `Sample` class) that can be invoked from the client code:

```
public static async void GetFlagValue()
{
    var sample = await InitializeAsync.Value;
    WriteLine($"The current flag value is {sample._flag}.");
}
```

## Demonstration 6

Here is a sample demonstration:

```
using static System.Console;

WriteLine("Examining async lazy Initialization");
Sample.GetFlagValue();
ReadKey();

class Sample
{
    private int _flag;
    private Sample(int flag)
    {
        _flag = flag;
    }
```

```
    private static Lazy<Task<Sample>> InitializeAsync = new(
       async () =>
       {
           int data = await Repository.GetDataAsync();
           return new Sample(data);
       }
     );
    public static async void GetFlagValue()
    {
        var sample = await InitializeAsync.Value;
        WriteLine($"The current flag value is {sample._flag}.");
    }
}

// The Repository class is not shown again to avoid repetition.
```

## Output

Here is some sample output:

```
Examining async lazy Initialization
The current flag value is 3.
```

## Using Nito.AsyncEx Library

If you read Stephen Cleary's blog at https://blog.stephencleary.com/2013/01/async-oop-2-constructors.html or Stephen Toub's article at https://devblogs.microsoft.com/pfxteam/asynclazyt/, you'll see the type `AsyncLazy<T>` that essentially simplifies the "appearance" of the code. By adding the `Nito.AsyncEx` namespace (I used version 5.1.2)

in my codebase, I can get such a type and write the equivalent code for InitializeAsync as follows (I prefixed the word *Nito* in this case so that you can compare InitializeAsync and NitoInitializeAsync easily):

```
private static AsyncLazy<Sample> NitoInitializeAsync = new(
  async () =>
  {
    int data = await Repository.GetDataAsync();
    return new Sample(data);
  }
);
```

In this case, I need to update the following method as follows( the old code is shown in comment for the comparison purposes):

```
public static async void GetFlagValue()
{
 //var sample = await InitializeAsync.Value;
 var sample = await NitoInitializeAsync;
 WriteLine($"The current flag value is {sample._flag}.");
}
```

With these changes, if you run the program, you'll see similar output once again.

---

**Note** Download the Demo6A_LazyInitializationAsyncUsingNito project from the Apress website to see the complete program.

---

# Introducing ValueTasks

You've been learning about tasks since the beginning of the book; they are fundamental to asynchronous programming. The `Task` and `Task<TResult>` classes help us a lot when we exercise asynchronous programming. From C# 5.0 onward we have the `async` and `await` keywords as well. While describing the C# 5.0 key features, the documentation at https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history#c-version-40 states the following:

> *… async and await are the real stars of this release.*

This chapter showed how these keywords made asynchronous programming easier.

One major improvement came in C# 7.0 where we saw the usage of `ValueTask` and `ValueTask<TResult>`. The `AsTask` method helps you retrieve a `Task` object from a `ValueTask` object. Obviously, an instance of `ValueTask<Result>` can be awaited as well.

The `Task` class is very flexible and provides lots of benefits. So, you may be wondering why we need the `ValueTask` instance.

In most cases, by invoking an asynchronous operation, you'll be awaiting for a resulting task. Just like you pay less for a product that has a limited number of features, you'd use `ValueTask` when you do not need all the features of the `Task` class.

Don't forget that `Task` is a class. So, it is heap-allocated. If you create lots of `Task` instances, you put extra pressure on the garbage collector (GC) for allocations and deallocations. It surely affects the performance of the application. In this context, you may note that `ValueTask` is of type `struct`, but not `class`.

Keeping the previous points in mind, you may try to implement a caching mechanism. However, let me tell you that it is not feasible to cache everything in every situation.

**POINT TO NOTE**

In this context, I suggest you read Microsoft's suggested article called "Understanding the Whys, Whats, and Whens of ValueTask" at https://devblogs.microsoft.com/dotnet/understanding-the-whys-whats-and-whens-of-valuetask/ to learn more about this.

## Consuming ValueTasks

Here is a synchronous method that returns a `ValueTask<int>`:

```
// Synchronous method
ValueTask<int> SomeMethod()
{
    // Some other code, if any
    return new ValueTask<int>(50);
}
```

You can consume this method as follows:

```
var task1 = SomeMethod();
WriteLine($"Received: {task1.Result}");
```

There is no surprise that you'll see the following output: `Received: 50`.

You can convert a `ValueTask<Result>` instance into a `Task<Result>` instance using `AsTask`. For example, the following code:

```
var task2 =
    SomeMethod()
    .AsTask()
    .ContinueWith
    (
```

```
        t => WriteLine($"The latest value is: {t.Result + 10}")
    );
task2.Wait();
```

can produce the following output: `The latest value is: 60`.

Let's consider an asynchronous method now. The following method returns a `ValueTask<int>` as well:

```
// Asynchronous method
async ValueTask<int> SomeMethodAsync()
{
    await Task.Delay(100);
    //return new ValueTask<int>(100); // Error now
    return 100; // OK
}
```

You can consume this method as follows:

```
int result = await SomeMethodAsync();
WriteLine($"Got: {result}");
```

which can produce the following output: `Got: 100`.

---

**Note** You can download the project `UnderstandingValueTask` to run these code segments and verify the output.

---

## Q&A Session

**Q6.9 Should I replace `Task` and `Task<TResult>` with `ValueTask` and `ValueTask<TResult>` in my asynchronous APIs?**

No. By default, you should always prefer `Task` or `Task<TResult>`. Unless you are really concerned about performance, `Task` or `Task<TResult>` is still preferred.

Also note that there are certain restrictions when you use `ValueTask` or `ValueTask<TResult>` in your programs; otherwise, the result will be undefined.

**Q6.10 What are the restrictions of using a `ValueTask` or `ValueTask<TResult>` in my programs?**

According to the documentation at https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask-1?view=net-8.0, if you perform any of the following operations, the result will be undefined:

- Awaiting the instance multiple times
- Calling `AsTask` multiple times
- Using `.Result` or `.GetAwaiter().GetResult()` when the operation hasn't yet completed, or using them multiple times
- Using more than one of these techniques to consume the instance

**Q6.11 When should I use `ValueTask` or `ValueTask<TResult>`?**

I suggest you follow Stephen Toub's suggestion. After a long discussion, he gave us the short rule as follows (see https://devblogs.microsoft.com/dotnet/understanding-the-whys-whats-and-whens-of-valuetask/):

> *With a ValueTask or a ValueTask<TResult>, you should either await it directly (optionally with .ConfigureAwait (false)) or call AsTask() on it directly, and then never use it again.*

# Reviewing Exceptions

Before you finish this chapter, I want to draw your attention to a special characteristic of an `await` expression. I first made the following class with two almost identical methods to demonstrate this. However, the first one is a normal method, and the second one is an `async` method. Let's look into the class:

```
class InvalidClass
{
    public static string UnknownUserDenied(string user)
    {
        throw new UnauthorizedAccessException($"{user} is
         unauthorized");
    }

    public static async Task<string> UnknownUserDeniedAsync
    (string user)
    {
        await Task.Delay(1000);
        throw new UnauthorizedAccessException($"{user} is
         unauthorized");
    }
}
```

Now let's execute the following code from the main thread:

```
using static System.Console;

WriteLine("Reviewing Exceptions.");

try
{
    var validateTask = Task.Run(() =>
```

```
      InvalidClass.UnknownUserDenied("abc"));
    WriteLine(validateTask.Result);
 }
catch (Exception e)
{
    WriteLine($"Exception type: {e.GetType()}, Message:
     {e.Message}");
}
```

Upon running this program, you'll see the following output:

```
Reviewing Exceptions.
Exception type: System.AggregateException, Message: One or more
 errors occurred. (abc is unauthorized)
```

I assume you are not surprised. You have seen a lot of discussion on `AggregateException` , and you know how to deal with it. For example, if you replace the existing `catch` block with the following one:

```
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        WriteLine($"Exception type: {e.GetType()}, Message:
         {e.Message}");
    }
}
```

you will see the specific exception(s) as follows:

```
Reviewing Exceptions.
Exception type: System.UnauthorizedAccessException, Message:
 abc is unauthorized
```

Now replace the previous `try-catch` block with the following one:

```
try
{
    var validateAsyncTask = await
       InvalidClass.UnknownUserDeniedAsync("abc");
}
catch (Exception e)
{
    WriteLine($"Exception type: {e.GetType()}, Message:
     {e.Message}");
}
```

Upon running this program, you'll see the following output:

```
Reviewing Exceptions.
Exception type: System.UnauthorizedAccessException, Message:
 abc is unauthorized
```

Notice that this time you are seeing the `UnauthorizedAccessException` directly. How is this possible? Let's see what Microsoft states about it. The documentation at https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/ states the following:

> *When code awaits a faulted task, the first exception in the AggregateException.InnerExceptions collection is rethrown.*

Now you understand that the `await` keyword has done this magic! It helps you pick the first exception inside the `AggregateException`.

---

**Note** Download the `Demo_ReviewingExceptions` project to see the complete program that contains the discussed code segments.

---

Congratulations! You have finished the final chapter of the book. It's time to check your understanding by attempting the following exercises.

**Author's Note:** Once I finished writing this book, I found that Stephen Toub and Scott Hanselman recently discussed some useful stuff at https://www.youtube.com/watch?v=R-z2Hv-7nxk. This is another good resource for learning about `async`/`await`.

# Exercises

Check your understanding by attempting the following exercises:

---

**Note** You can assume that in addition to the default settings, the statement: `using static System.Console;` has also been added at the beginning of all the following exercises.

---

**E6.1** Can you predict the output of the following code?

```
var task1 = Task.Run(
    async () =>
    {
       await Task.Delay(1000);
       return "Hello";
    });
WriteLine(task1.Result);
```

**E6.2** If you replace `Task.Run` with `Task.Factory.StartNew` in the previous code, will you see the same output?

**E6.3** Can you predict the output of the following code?

```
var task3A = new ValueTask<int>(10);

var task3B = task3A.ContinueWith(
   (t) =>
```

```
    {
        int temp = task3A.Result;
        return ++temp;
    }
);
WriteLine($"The final result is: {task3B.Result}");
```

**E6.4** Can you predict the output of the following code?

```
string result = await GetWeatherReportAsync();
WriteLine(result);

async ValueTask<string> GetWeatherReportAsync()
{
    await Task.Delay(100);
    return "This is a beautiful day!";
}
```

**E6.5** Can you predict the output of the following code?

```
var number= await GetNumberAsync().AsTask();
WriteLine($"Got the number: {number}");

async ValueTask<int> GetNumberAsync()
{
    await Task.Delay(100);
    return new ValueTask<int>(50);
}
```

**E6.6** Can you compile the following code?

```
var task6 = Task.Run(
    async () =>
    {
        await Task.Delay(1000);
```

```
        return new ValueTask<int>(10);
    });
WriteLine($"Task 6 returns: {task6.GetAwaiter().GetResult()}");
```

**E6.7** Can you compile the following code?

```
var task7 = Task.Run(
    async () =>
    {
        await Task.Delay(1000);
        return new ValueTask<int>(20).AsTask();
    });
WriteLine($"Task 7 returns: {await task7.GetAwaiter().
 GetResult()}");
```

**E6.8** Can you predict the output of the following code?

```
var task8 =
    Task.Factory.StartNew(
    async () =>
    {
        await Task.Delay(1000);
        return Task.Run(()=>"Hello");
    });
WriteLine($"Task 8 returns:{await await task8.Result}");
```

**E6.9** Demonstration 1 showed you an asynchronous program using the `async` and `await` keywords. Can you write an equivalent program using a task-based continuation?

**E6.10:** Write a program that requests the content of a web page. Ensure that your program demonstrates the power of the `async` and `await` keywords.

**Hint**: Write an `async` function that accepts a URL that can be passed to the `GetStringAsync` method. Before you print the content, ensure that the calling thread is not blocked while the function is working. You may note that starting in .NET 6, the `WebClient` class is deprecated. So, try using the `HttpClient` class in this program.

**E6.11** Add the cancellation capability to the previous program (E6.10).

**E6.12** Can you summarize the benefits of using the `async` and `await` keywords?

# Summary

The `Task` class and the derived `Task<TResult>` were introduced to make asynchronous programming easy. Still, there were challenges. To overcome those challenges and make things simpler, C# 5 introduced the `async` and `await` keywords, and C# 7.0 came up with `ValueTask` and `ValueTask<TResult>`. These are integral parts of asynchronous programming nowadays. This chapter discussed them using various code segments and complete programs. Upon completion of this chapter, you should not find difficulties in answering the following questions:

- How do the `async` and `await` keywords make asynchronous programming easy?
- What are the restrictions associated with the `async` and `await` keywords?
- How does the `Task.Factory.StartNew` and `Task.Run` methods treat ordinary lambdas and asynchronous lambdas?
- How should you perform asynchronous initializations?
- Why `ValueTasks` were introduced? How do they help?

- How can you consume the `ValueTasks`?
- What are the restrictions associated with the `ValueTasks`?

# Solutions to Exercises

Here is a sample solution set for the exercises in this chapter.

## E6.1

You should see the following output: `Hello`.

## E6.2

No. This time you'll see the following:

```
System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1+AsyncS
tateMachineBox`1[System.String,Program+<>c+<<<Main>$>b__0_1>d]
```

**Clue:** You have seen that `Task.Factory.StartNew` treats async lambdas differently. To see the output `Hello`, you can either use the `Unwrap` method as follows:

```
var task2 = Task.Factory.StartNew(
    async () =>
    {
        await Task.Delay(1000);
        return "Hello";
    }).Unwrap();
WriteLine(task2.Result);
```

or, use one more `await` keyword as follows:

```
var task2 = await Task.Factory.StartNew(
    async () =>
    {
        await Task.Delay(1000);
        return "Hello";
    });
WriteLine(task2.Result);
```

# E6.3

This program will not compile. You will see the following error:

```
CS1061 'ValueTask<int>' does not contain a definition
for 'ContinueWith' and no accessible extension method
'ContinueWith' accepting a first argument of type
'ValueTask<int>' could be found (are you missing a using
directive or an assembly reference?)
```

**Additional Note:** You can replace the line:

```
var task3B = task3A.ContinueWith(  // Error CS1061
```

with the following line that uses the `AsTask` method:

```
var task3B = task3A.AsTask().ContinueWith(  //OK
```

and get output as follows:

```
The final result is: 11
```

# E6.4

You should see the following output: `This is a beautiful day!`

# E6.5

This code will not compile because the method implementation is incorrect. Here is some sample output for your reference:

```
CS4016 Since this is an async method, the return expression
must be of type 'int' rather than 'ValueTask<int>'
```

So, if you replace the existing method with the following method:

```
async ValueTask<int> GetNumberAsync()
{
    await Task.Delay(100);
    //return new ValueTask<int>(50); // Error CS4016
    return 50; // OK
}
```

you can see the following output:

```
Got the number: 50
```

# E6.6

Yes. Notice that this time you are passing the `async` lambda inside the `Task.Run` method and `task6` is of type `Task<ValueTask<int>>`. This code will produce the following output:

```
Task 6 returns: 10
```

You may also note that if you write something like the following:

```
var task6 = Task.Run(
    async () =>
    {
        await Task.Delay(1000);
        return 10; // OK too
        //return new ValueTask<int>(10); // OK
    });
```

you'll see the same output, but this time `task6` is a `Task<int>`.

## E6.7

Yes. This code will produce the following output:

```
Task 7 returns: 20
```

## E6.8

This time `task8` is of type `Task<Task<Task<string>>>`. So, do not be surprised about the two `await` keywords in the following expression: `await await task8.Result.` By using them, you can unwrap the type `Task<Task<Task<string>>>` twice to get the output:

```
Task 8 returns: Hello
```

## E6.9

The following program shows you a task-based continuation example. It is similar to Demonstration 1. You may note that, in this case, you observe a slightly different output where the line "`The method resumes.`" appears before the line "`The main thread continues.`". Why? In this program, I'm synchronously waiting for the continuation task to complete.

However, in Demonstration 1, the control returned to the calling thread as soon as it hit the first `await` expression (and it did not wait for the method to be completed). For an easy comparison, I have kept both code segments and the associated output.

```
using static System.Console;

WriteLine("The main thread starts.");

WriteLine("-------------------------------------");
```

```
#region Calling the asynchronous code (Used in  demo 1)
Task<int> getValue = GetHundredAsync();
WriteLine("The main thread continues.");
// Consuming the return value which is an int
int result1 = await getValue;
WriteLine($"The invoked task returns: {result1}");
#endregion
WriteLine("-------------------------------------");

#region Task-based asynchronous equivalent
Task<int> result = GetHundred();
WriteLine("The main thread continues.");
var remainingTask = result.ContinueWith(
    t => WriteLine($"The invoked task returns: {t.Result}")
);
remainingTask.Wait();
#endregion

WriteLine("-------------------------------------");

WriteLine("The main thread ends.");

// Asynchronous version
static async Task<int> GetHundredAsync()
{
    WriteLine("The method is arranging the number.");
    await Task.Delay(1500);
    WriteLine("The method resumes.");
    return 100;
}

// Synchronous version
static Task<int> GetHundred()
```

```
{
    WriteLine("The method is arranging the number.");
    Task.Delay(1500);
    WriteLine("The method resumes.");
    return Task.Run(() => 100);
}
```

This program produces the following output:

```
The main thread starts.
-----------------------------------
The method is arranging the number.
The main thread continues.
The method resumes.
The invoked task returns: 100
-----------------------------------
The method is arranging the number.
The method resumes.
The main thread continues.
The invoked task returns: 100
-----------------------------------
The main thread ends.
```

## E6.10

Here is a sample program where I arbitrarily chose the URL https://www.aboutamazon.in/what-we-do/amazon-pay:

```
using static System.Console;

WriteLine("Exercise 6.10");
string url = "https://www.aboutamazon.in/what-we-do/
 amazon-pay";
```

```
var webContent = PrintContents(url);
WriteLine("Shortly, you'll see the content...");
WriteLine(webContent.Result);
ReadKey();

static async Task<string> PrintContents(string url)
{
    var client = new HttpClient();
    var contents = await client.GetStringAsync(url);
    return contents;
}
```

Since the output is long, I've condensed it (I highlighted a portion in bold so that you can verify the content):

```
Exercise 6.10
Shortly, you'll see the content...
<!DOCTYPE html>
<html class="PageDefault" lang="en">

// Intermediate output skipped

    <div class="HubPage1-content">

    <div class="HubPage1-description">
            Amazon Pay simplifies daily life and fulfills
            the aspirations of every Indian by solving their
            payment and financial needs. It offers a trusted,
            convenient, and rewarding way for millions of cash
            customers to make digital payments, both on Amazon
            and at offline stores. It is also used by millions
            of merchants every day to provide their customers
            with a simple, safe, and reliable method to pay.
    </div>
```

```
    </div> <!-- END HubPage1-content -->
</div> <!-- END HubPage1-container -->

// Remaining output is skipped too
```

# E6.11

I leave this exercise to you.

**Hint:** This time consider using an overloaded version of `GetStringAsync` that accepts both the URL and the `CancellationToken` as parameters.

# E6.12

The following list includes some of the key benefits of this:

- You write less code.
- You do not need continuations that are based on anonymous callbacks.
- The asynchronous code that uses `async` and `await` keywords looks very similar to its counterpart, i.e., the synchronous version.
- The exception handling becomes easier.

## APPENDIX A

# Supplementary Notes

A *multicore* machine can execute multiple threads simultaneously. With the advancement of technology, most computers have multicore processors nowadays. To make fast-paced applications, you can take advantage of the hardware and parallelize your code by distributing the work among these cores. Throughout this book, you explored the possibilities using various C# constructs. This appendix includes some additional information, useful suggestions, and supplementary materials for you.

## PFX Overview

When you jump into parallel programming in C#, you will often hear the acronym PFX. This stands for the Parallel Framework (often called Parallel Framework Extensions) in C#.

You may be wondering why didn't I mention PFX earlier. This is because PFX includes lots of constructs, and I didn't want to overwhelm you. Since you have completed Chapters 1–6, you can realize that you've learned most of these components.

## Data Structures

At `https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/data-structures-for-parallel-programming`, Microsoft talks briefly about the following data structures that are useful for parallel programming.

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5

- Concurrent collection classes
- Lightweight synchronization primitives such as `SemaphoreSlim`, `CountdownEvent`, `ManualResetEventSlim`, `Barrier`, `SpinLock`, and `SpinWait`
- Types for lazy initialization such as `System.Lazy<T>`, `System.Threading.ThreadLocal<T>`, and `System.Threading.LazyInitializer`
- Aggregate exceptions (`System.AggregateException` type)

You learned about concurrent collections in Chapter 3, `Lazy<T>` in Chapter 6, and `AggregateException` in Chapter 2. This means we have already covered many of these components. Though the book does not try to cover all of them, this appendix will cover a few more synchronization primitives such as `SemaphoreSlim`, `CountdownEvent`, and `ManualResetEventSlim`.

## PFX Components

When .NET Framework 4.0 was released, Microsoft stated the following (https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/september/what%E2%80%99s-new-in-the-base-class-libraries-in-net-framework-4):

> *Another major new addition to the BCL in .NET 4 is the Parallel Extensions (PFX) feature that is being delivered by the Parallel Computing Platform team. PFX include the Task Parallel Library (TPL), Coordination Data Structures, Concurrent Collections and Parallel LINQ (PLINQ)—all of which will make it easier to write code that can take advantage of multicore machines.*

Since the task parallelism constructs and the `Parallel` class are known together as the Task Parallel Library (TPL), you can say that PFX essentially encompasses the following:

- The task parallelism constructs (you learned about many of these in Chapter 1 and Chapter 2)
- The `Parallel` class (you learned about this class in Chapter 3)
- The coordination data structures including concurrent collections (you learned these structures in Chapter 2 and Chapter 3)
- Parallel LINQ (PLINQ) (you learned about this in Chapter 5)

## How Does PFX Help?

At a high level, parallel programming involves the following steps:

1. Partitioning the "big" work into "smaller works"
2. Exercising those "small works" in parallel
3. Collecting and organizing the results properly

This can be done using classic multithreading constructs, but things are not straightforward. The PFX libraries were designed to make things easier for you.

## Partitioning Strategies

While exercising parallelism, there are two partitioning strategies:

- Partitioning the data (data parallelism)
- Partitioning the tasks (task parallelism)

As the name implies, in the case of data parallelism, the source collection is partitioned so that different threads can work on those segments concurrently. TPL supports this concept through the `Parallel` class. More specifically, PFX provides a basic form of structured parallelism via `Parallel.For`, `Parallel.ForEach`, and `Parallel.Invoke`.

However, in case of task parallelism, the independent tasks run concurrently.

**POINT TO NOTE**

Data parallelism tends to be *structured* because the parallel work units start and finish in the same place in the program. On the contrary, task parallelism tends to be *unstructured*, because the parallel work units may start and finish in different places.

## Q&A Session

**QApp1.1 My understanding is that the `Parallel` class helps me to segregate the work but I need to gather the result. However, in the case of task parallelism, I need to partition the work too. Is that correct?**

I appreciate your observation. You are correct.

# More on Synchronization Constructs

In Chapter 3 we divided the synchronization constructs into four categories and discussed one from each category. Here you'll see a few more useful constructs. You can use them as needed.

## Signaling Using ManualResetEvent

You already learned signaling using `AutoResetEvent`. Demonstration 4 in Chapter 3 showed you a sample implementation as well. This time you'll learn about `ManualResetEvent`. I also showed you in Chapter 3 that when I replaced the line `var resetEvent = new AutoResetEvent(false);` with `var resetEvent = new ManualResetEvent(false);` I got the following output:

```
Two customers are approaching the mall.
The customer 1 is waiting for the entry pass.
The customer 2 is waiting for the entry pass.
Press any key to issue the signal from the main thread.
The customer 1 enters the mall.
The customer 2 enters the mall.
Press any key to issue another signal from the main thread.
Another customer is approaching the mall.
Press any key to exit.
The customer 3 is waiting for the entry pass.
The customer 3 enters the mall.
```

Then I pointed out the difference between the outputs. In this output, both customer 1 and customer 2 were able to enter the mall when the main thread issued the signal. In addition, customer 3 was also able to enter the mall because, in between, the gate was not closed.

Now the question is: how to close the gate? You need to call the `Reset` method. Let's look at the following program.

## Demonstration 1

Here is a sample implementation for you:

```
using static System.Console;

var resetEvent = new ManualResetEvent(false);
```

```
WriteLine("Two customers are approaching the mall.");
Task.Run(VisitMall);
Task.Run(VisitMall);

// Imposing some delay to mimic a real-world scenario
Thread.Sleep(2000);

WriteLine("Press any key to issue the signal from the main
 thread.");
ReadKey();
resetEvent.Set();
Thread.Sleep(1000);

// Reset will restrict the new task to continue
WriteLine("Resetting the signal now.");
resetEvent.Reset();

WriteLine("Another customer is approaching the mall.");
Task.Run(VisitMall);

WriteLine("Press any key to exit.");
ReadKey();

void VisitMall()
{
    // Imposing a small delay to mimic a real-world scenario
    Thread.Sleep(1000);
    WriteLine($"The customer {Task.CurrentId} is waiting for
     the entry pass.");
    resetEvent.WaitOne();
    WriteLine($"The customer {Task.CurrentId} enters the mall.");
}
```

## Output

Here is a sample output (I pressed the Enter key to issue the signals):

```
Two customers are approaching the mall.
The customer 1 is waiting for the entry pass.
The customer 2 is waiting for the entry pass.
Press any key to issue the signal from the main thread.
The customer 1 enters the mall.
The customer 2 enters the mall.
Resetting the signal now.
Another customer is approaching the mall.
Press any key to exit.
The customer 3 is waiting for the entry pass.
```

## Analysis

You can see that when the gate opened, customer 1 and customer 2 both entered the mall. However, the gate was closed shortly after that. This is why customer 3 could not enter the mall.

## Q&A Session

**QApp1.2 While discussing `AutoResetEvent`, you talked about a real-world scenario. How can I relate a real-world scenario to `ManualResetEvent`?**

In my city, when I go to the cinema, I usually see an entire family or group of people enter the movie hall with a single ticket. It means that a group of people can pass the entry gate together. Look at the previous demonstration now. You can see that when the main thread issued the signal, the waiting customers entered the mall together as well.

# Signaling Using ManualResetEventSlim

`ManualResetEventSlim` is a lightweight alternative to `ManualResetEvent`. Let's make the following changes in the previous demonstration:

- Replace the line `var resetEvent = new ManualResetEvent(false);` with `var resetEvent = new ManualResetEventSlim(false);`.
- This time replace the `WaitOne` method with the `Wait` method inside the `VisitMall` method. This `Wait` method blocks the current thread until the current `ManualResetEventSlim` is set. This change is required because this slim version does not subclass `WaitHandle`.

## Demonstration 2

Since there are only two minor changes, I am not showing you the complete program here. You can download the `Demo2_ManualResetEventSlim` project from the Apress website to see the complete program and execute it.

## Output

Once you make these changes, you'll see similar output to what was shown in the previous demonstration.

## Q&A Session

**QApp1.3 When should I prefer `ManualResetEventSlim` over `ManualResetEvent` (and vice versa)?**

You should prefer `ManualResetEventSlim` when you expect a short waiting time. However, you may note that though the lightweight

synchronization types normally provide better performance, they cannot be used for interprocess synchronization. In other words, you can use them in single applications.

You can find additional information at https://learn.microsoft.com/en-us/dotnet/api/system.threading.manualreseteventslim?view=net-8.0.

> *ManualResetEventSlim uses busy spinning for a short time while it waits for the event to become signaled. When wait times are short, spinning can be much less expensive than waiting by using wait handles. However, if the event does not become signaled within a certain period of time, ManualResetEventSlim resorts to a regular event handle wait.*

## Signaling Using CountdownEvent

Using `ManualResetEvent`, you make one thread superior. Here one thread can release several blocked threads. `CountdownEvent` can be used for a reverse scenario where multiple threads collectively release one blocked thread. In other words, a thread is unblocked when it receives signals from multiple threads.

To show you an example, let me talk about our metro railway stations one more time. I have told you that only one passenger can pass through a gate at a time, but I did not mention that there are multiple gates through which passengers can enter or exit. During the festive seasons or office hours, the metro officials open those gates to control the crowd. However, an ordinary shop may not have this option. To control the crowd, they may need to restrict the entries.

Suppose that a well-known company has announced that its next-generation product is about to be launched on a particular day. Let's assume that on the day of the release, customers are approaching a local shop, which is not very big! Let's further assume that initially, the

gatekeeper allows the customers. However, to control the crowd, the gatekeeper needs to close the gate shortly. He decides to open the gate for a new customer if at least three current customers exit from the shop. The upcoming program demonstrates this situation. Here are the key characteristics of this program:

- The main thread (gatekeeper) waits to receive signals from three worker threads (customers). It blocks itself by calling the `Wait` method.
- Each of these worker threads issues a signal by invoking the `Signal` method.

## Demonstration 3

Here is the demonstration:

```
using static System.Console;

var countdownEvent = new CountdownEvent(3);

WriteLine("Five customers are trying to enter the mall.");
for (int i = 0; i < 5; i++)
{
    Task.Run(VisitMall);
}
WriteLine($"The gatekeeper waits for {countdownEvent.
 CurrentCount} customers to exit.");
countdownEvent.Wait();
WriteLine($"The gatekeeper got {countdownEvent.InitialCount}
 signals.");
WriteLine("A new customer can enter the mall now.");
WriteLine("Press any key to exit.");
ReadKey();
```

```
void VisitMall()
{
    int random = new Random().Next(1, 5);
    Thread.Sleep(1000);
    WriteLine($"The customer {Task.CurrentId} starts
     purchasing.");
    Thread.Sleep(random* 500);
    WriteLine($"-----The customer {Task.CurrentId} is
     exiting now.");
    countdownEvent.Signal();
}
```

## Output

Here is some sample output:

```
Five customers are trying to enter the mall.
The gatekeeper waits for 3 customers to exit.
The customer 3 starts purchasing.
The customer 2 starts purchasing.
The customer 1 starts purchasing.
The customer 4 starts purchasing.
-----The customer 3 is exiting now.
The customer 5 starts purchasing.
-----The customer 1 is exiting now.
-----The customer 4 is exiting now.
The gatekeeper got 3 signals.
A new customer can enter the mall now.
Press any key to exit.
-----The customer 2 is exiting now.
-----The customer 5 is exiting now.
```

## Additional Note

The `CountdownEvent` class has many other useful methods with their usual meaning. For example, you can use the `AddCount` method to increment the `CountdownEvents` current count by one. You can also use the overloaded version `AddCount(int signalCount)` to increment the current count by a specified number. However, while using the `AddCount` method, you need to be careful; if you use this method when the `CountdownEvents` current count is zero, you will observe the following error message:

```
System.InvalidOperationException: 'The event is already
signaled and cannot be incremented.'
```

Therefore, you can consider using the `TryAddCount` method in such a context. I suggest you experiment with the other methods as well.

**POINT TO NOTE**

Ideally, you should try to release the resources whenever possible. The same rule applies when you use the `ManualResetEventSlim` class or the `CountdownEvent` class. However, I do not discuss garbage collection in this book. To focus on the key areas of the discussion and make the code short and simple, I ignored this suggestion.

# Using Mutex

A mutex is another useful synchronization primitive; it is similar to the `lock` statements. The documentation at https://learn.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=net-8.0 states the following:

> *Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one thread. If a thread acquires a mutex, the second thread that wants to acquire that mutex is suspended until the first thread releases the mutex.*

Demonstration 1 in Chapter 3 showed you an erroneous program. We corrected the program by modifying the `Account` class in Demonstration 2 of that chapter. To ensure synchronized access, we used `lock` statements in the modified version of the `Account` class. This time I'll show you another program that uses a different version of the `Account` class using the `Mutex` class.

## Demonstration 4

Let's understand the changes in this new version of the `Account` class step-by-step.

- The `Account` class has a `Mutex` object. It also contains a Boolean variable, called `_lockBalance`.
- Inside the `Credit` method, the `_lockBalance` variable is used as follows: `_lockBalance = _mutex.WaitOne();` Using this line, you try to block the current thread until the current `WaitHandle` receives a signal. If the method returns `true`, you can perform the intended operation. In simple words, you block the current thread until you get ownership of the mutex. Otherwise, you cannot operate.

**POINT TO NOTE**

A calling thread can be unblocked in any of the following cases:

**Case 1**: When the `WaitOne` method returns true. This means that no one owns the mutex at that moment.

**Case 2**: When a specified time limit is elapsed. You can specify this time-out interval to an overloaded version of the `WaitOne` method.

- If you get the ownership, after performing the intended job, you must release the mutex by calling the `ReleaseMutex` method. Similar to a lock statement, a mutex is released only from the thread that acquired the lock. Now you understand that the `WaitOne` method is used to lock and `ReleaseMutex` is used to unlock.
- The `Debit` method is similar. However, as per the name, it debits from the balance.

Since there is no change in the client code and the namespace, I am not showing them again to avoid repetition. You can download the project `Demo4_Mutex` to execute the complete program. In that program, I also kept all the different variations of the `Account` class in the commented code. You already saw these variations in other examples, but I'm showing them again for reference.

I assume you will not have any difficulty understanding the following code that shows the modified version of the `Account` class:

```
#region Using mutex
class Account
{
    public decimal Balance { get; set; }
    private readonly Mutex _mutex = new();
    private bool _lockBalance;
    public void Credit(decimal amount)
    {
        _lockBalance = _mutex.WaitOne();
        if (_lockBalance)
        {
            try
            {
```

```
            Balance += amount;
            WriteLine($"The balance after the credit of
             ${amount} is ${Balance}");
        }
        finally
        {
            if (_lockBalance) _mutex.ReleaseMutex();
        }
    }
    else
    {
        WriteLine($"Could not credit: {amount}");
    }
}
public void Debit(decimal amount)
{
    _lockBalance = _mutex.WaitOne();
    if (_lockBalance)
    {
        try
        {

            Balance -= amount;
            WriteLine($"The balance after the debit of
             ${amount} is ${Balance}");
        }
        finally
        {
            if (_lockBalance) _mutex.ReleaseMutex();
        }
    }
```

```
        else
        {
            WriteLine($"Could not debit: {amount}");
        }
    }
}
#endregion
```

## Output

Here is some sample output:

```
The balance after the credit of $200 is $200
The balance after the debit of $500 is $-300
The balance after the credit of $100 is $-200
The balance after the credit of $300 is $100
The balance after the debit of $100 is $0
The current balance is $0
```

## Additional Note

The `Mutex` class is a powerful synchronization primitive. You can use it for interprocess communications as well. I recommend that you read the documentation at https://learn.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=net-8.0 to learn more about the `Mutex` class.

## Q&A Session

**QApp1.4 Can you show me an example where a mutex is used for interprocess communication?**

As suggested, if you read the documentation at the previous link, you'll see that mutexes can be any of the following types:

- Unnamed local mutexes
- Named system mutexes

For the interprocess communication, you'll use a named system mutex that can be created by providing a name to the constructor. (You can learn more about the "naming" from the previously mentioned link.)

**Let's look at a sample program where you'd like to ensure only one instance of a program can run at a time**. Before executing the program, here is the flow:

- First, the `OpenExisting` method tries to open the specified named mutex. If the mutex does not exist, it'll throw an exception, named `WaitHandleCannotBeOpenedException`.
- If you encounter the exception, you understand that mutex does not exist; so, you create a mutex inside the `catch` block.
- Once the mutex is available, you try to get it before launching the application.

The remaining code should be easy for you because you're already familiar with the `Waitone` and the `ReleaseMutex` methods. However, this time I passed 5 sec as a parameter to the `WaitOne` method to avoid an indefinite waiting time. Here is the complete program:

```
using static System.Console;

const string appName = "MyApp";
Mutex mutex;
try
{
    mutex = Mutex.OpenExisting(appName);
    // If the previous line succeeds, we can run the following line.
```

```
    WriteLine($"An instance of the application {appName} is
     already running!");
    return;
}
catch (WaitHandleCannotBeOpenedException)
{
  // Since the mutex is unavailable, we can create it
    mutex = new(false, appName);
}

// The mutex is already created. Trying to get it before
// launching the app.

bool getMutex = mutex.WaitOne(5000);
if (getMutex)
{
    try
    {
        // Launching the application now
        LaunchApplication();
    }
    finally
    {
        WriteLine($"Press the 'Enter' key to close the app.");
        ReadKey();
        mutex.ReleaseMutex();
    }
}
```

```
void LaunchApplication()
{
    WriteLine($"Starting the {appName} application");
    // Do something, if needed
}
```

Once you run this program, you'll see the following output:

```
Starting the MyApp application
Press the 'Enter' key to close the app.
```

**Do not press the Enter key now. Instead, try to launch the same program from the command line**. This time, you'll see the screen in Figure App-1.

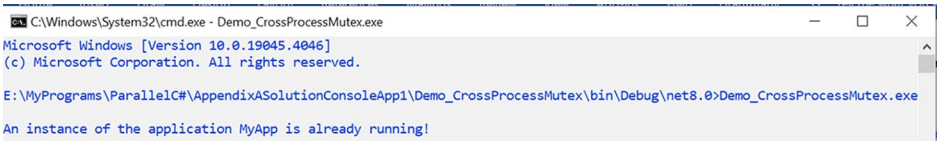

C:\Windows\System32\cmd.exe - Demo_CrossProcessMutex.exe
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.
E:\MyPrograms\ParallelC#\AppendixASolutionConsoleApp1\Demo_CrossProcessMutex\bin\Debug\net8.0>Demo_CrossProcessMutex.exe
An instance of the application MyApp is already running!

***Figure App-1.*** *Trying to launch the same application from the command line*

You can also use multiple command prompts to verify the scenario. I believe that you have got the idea!

**Author's Note:** If you first launch the application from the command prompt, you should not try to run the application from Visual Studio again. Instead, you can launch another command prompt to verify the scenario. Figure App-2 shows an example.

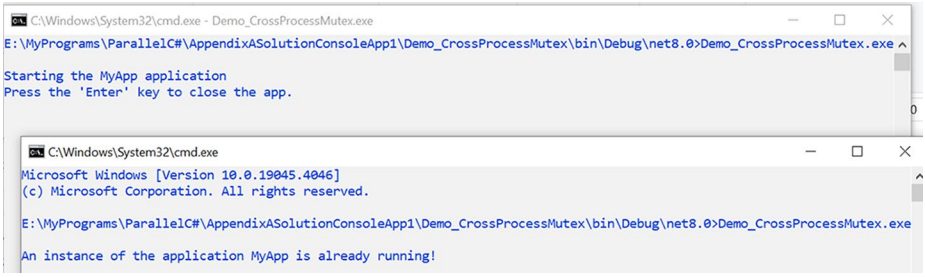

C:\Windows\System32\cmd.exe - Demo_CrossProcessMutex.exe
E:\MyPrograms\ParallelC#\AppendixASolutionConsoleApp1\Demo_CrossProcessMutex\bin\Debug\net8.0>Demo_CrossProcessMutex.exe
Starting the MyApp application
Press the 'Enter' key to close the app.
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.
E:\MyPrograms\ParallelC#\AppendixASolutionConsoleApp1\Demo_CrossProcessMutex\bin\Debug\net8.0>Demo_CrossProcessMutex.exe
An instance of the application MyApp is already running!

***Figure App-2.*** *Trying to launch the same application from two different command prompts*

You can download the `Demo_CrossProcessMutex` project from the Apress website to execute this program.

## Using SemaphoreSlim

Consider a typical appointment booking scenario in a doctor's office or a hospital. At any time, different numbers of patients can try to connect to the office over the phone to book an appointment. However, you know that there are a limited number of help-desk members who can pick up the calls to schedule an appointment. To demonstrate an example, let's assume that there are only two such members who can answer the incoming calls. As a result, at any time a maximum of two patients' appointment booking procedures can happen, and other callers need to wait to get their turn. The use of semaphores perfectly suits this scenario.

In this context, you can use either the `Semaphore` or `SemaphoreSlim` class. As per the name, the latter one is a slim and optimized version of the first. However, this slim version cannot be used if you want to handle interprocess communications. In the upcoming example, I'm going to use the optimized version, i.e., `SemaphoreSlim`.

How does the semaphore work? See the following points:

- A semaphore's count denotes the number of spaces (in our example, the available help-desk members). This count is increased when a thread releases the semaphore and is decreased when a thread enters the semaphore.
- Microsoft ensures (see https://learn.microsoft.com/en-us/dotnet/standard/threading/semaphore-and-semaphoreslim) that there is no guaranteed order, such as last-in, first-out (LIFO) or first-in, first-out (FIFO) in which blocked threads enter the semaphore.
- The `Release` method is used to release the semaphore. Both the `Semaphore` and `SemaphoreSlim` classes have the `Release` method.
- While using the `Semaphore` class, by calling the `WaitOne` method (this is inherited from the `WaitHandle` class), you can block the current thread until the current `WaitHandle` receives a signal. However, I am using the `SemaphoreSlim` class in the upcoming program; so, you'll see me using the `Wait` method (it also blocks the current thread until it enters the semaphore).

Let us understand the code now. The `SemaphoreSlim` class has the following constructors:

```
public SemaphoreSlim(int initialCount);
public SemaphoreSlim(int initialCount, int maxCount);
```

The `initialCount` parameter is used to indicate the initial number of requests for the semaphore that can be granted concurrently. The other version is used when you set the initial count along with the maximum count that can be granted.

Let's assume the patients start calling for the appointments before the help-desk members have arrived. They eagerly wait for someone to answer their call or get an appointment. To represent this situation, I create the following semaphore with a maximum count of two and an initial count of zero. (Since in our case two help-desk members can book the appointment, I set the maximum count to two.)

```
SemaphoreSlim semSlim = new(0, 2);
```

Once the help-desk members are available, the booking procedure starts. Now two people can schedule the appointment. So, you'll see the following code segment with supporting messages for your reference:

```
// Restore the Slimaphore count to its maximum value.
WriteLine("The helpdesk members have arrived and the booking
  procedure has started. ");
semSlim.Release(2);
helpdeskMemberCount = semSlim.CurrentCount;
```

The remaining part is easy. Let's see a sample demonstration that uses the `SemaphoreSlim` class.

## Demonstration 5

Here is the complete program:

```
using static System.Console;

SemaphoreSlim semSlim = new(0, 2);

for (int i = 1; i <= 5; i++)
{
    _ = Task.Run(() => Appointment.BookAppointment(semSlim));
}
```

```
// Let the patients wait
Thread.Sleep(2000);
int helpdeskMemberCount = semSlim.CurrentCount;
WriteLine($"Currently, {helpdeskMemberCount} helpdesk member(s)
 are available.");
WriteLine("Please wait...");
// Restore the Slimaphore count to its maximum value.
WriteLine("The helpdesk members have arrived and the booking
 procedure has started. ");
semSlim.Release(2);
helpdeskMemberCount = semSlim.CurrentCount;

ReadKey();

class Appointment
{
  public static void BookAppointment(SemaphoreSlim sem)
   {
    try
    {
     int patientId = Task.CurrentId.HasValue ? (int)Task.
      CurrentId : 0;
     WriteLine($"Patient: {patientId} calls for an appointment
      booking.");
     sem.Wait();
     WriteLine($"**Patient: {patientId} is getting the
      appointment. **");
     // The booking time for different patients can be different
     Thread.Sleep(patientId * 500);
     WriteLine($"\tPatient: {patientId} disconnects the call.");
     sem.Release();
    }
```

```
    catch (Exception e)
    {
     WriteLine($"Something went wrong: {e.Message}");
    }
   }
}
```

## Output

Here is some sample output:

```
Patient: 2 calls for an appointment booking.
Patient: 4 calls for an appointment booking.
Patient: 1 calls for an appointment booking.
Patient: 3 calls for an appointment booking.
Patient: 5 calls for an appointment booking.
Currently, 0 helpdesk member(s) are available.
Please wait...
The helpdesk members have arrived and the booking procedure has
started.
**Patient: 2 is getting the appointment. **
**Patient: 4 is getting the appointment. **
        Patient: 2 disconnects the call.
**Patient: 1 is getting the appointment. **
        Patient: 1 disconnects the call.
**Patient: 3 is getting the appointment. **
        Patient: 4 disconnects the call.
**Patient: 5 is getting the appointment. **
        Patient: 3 disconnects the call.
        Patient: 5 disconnects the call.
```

## Q&A Session

**QApp1.5 It appears to me that a semaphore with an initial count of 1 is similar to a lock (or mutex). Is this correct?**

Yes, these look similar. However, you should remember that the thread that obtained the lock is the only one that can release the lock. However, a semaphore does not have an owner; any thread can call `Release` on it.

# Avoiding Deadlocks

Up until now, you have seen many synchronization primitives. I assume that you understand the async programming model as well. I recommend that you read the online documentation at https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios to get some additional advice from Microsoft. Its suggestion is that you write code that awaits tasks in a nonblocking manner. A blocking call eventually may lead to a deadlock situation. This is why Microsoft suggests that you do the following:

- Use `await` instead of `Task.Wait` or `Task.Result` when you want to retrieve the result of a background task.
- Use `await Task.WhenAny` instead of `Task.WaitAny` when you want to wait for a task to complete.
- Use `await Task.WhenAll` instead of `Task.WaitAll` when you want to wait for all tasks to complete.
- Use `Task.Delay` instead of `Thread.Sleep`.

In this book, you started experimenting with parallel programming using tasks. So, I needed to use the constructs like `Task.Wait`, `Task.WaitAny`, `Task.WaitAll`, or `Task.Result` in my demonstrations. In a simple application, finding a deadlock is relatively easy. So, you do not need to worry much. However, for an enterprise edition, it's better to keep these suggestions in mind.

# More on Async/Await

In earlier days, you'd see the use of asynchronous programming models (also known as APMs or `IAsyncResult` patterns) and event-based asynchronous patterns (EAPs). I also discussed them in my book *Design Patterns in C# 2nd Edition* (Apress, 2020). However, these are considered obsolete patterns nowadays. This is why this book does not discuss them.

What are the alternatives? Microsoft says the following (see https://learn.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap):

> *In .NET, the task-based asynchronous pattern is the recommended asynchronous design pattern for new development.*

This is why you are about to see the discussions on TAP.

## Task-Based Asynchronous Patterns

To qualify for a task-based asynchronous pattern (TAP) pattern, Microsoft has detailed guidelines (see https://learn.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap). However, for a simple application, you do not have to worry about all those details. For your quick reference, I have summarized the important ones here:

- TAP is based on `Task` and `Task<Result>`. So, the asynchronous methods inside TAP return awaitable types such as `Task`, `Task<TResult>`, `ValueTask`, and `ValueTask<TResult>`.
- If you have an asynchronous method that does not return awaitable types, you can use names like `StartXXX`, `BeginXXX`, or any other verb.

- You already know that the asynchronous methods normally have the `Async` suffix. You should maintain the same convention when you implement the TAP pattern.
- However, there are a few exceptions to the previous rule, such as when you use combinators such as `WhenAll` or `WhenAny`.
- Consumers of a TAP method always work with hot running tasks. It means that they never call the `Start` method on them.
- The return type of a TAP method depends on the corresponding synchronous method. For example, if a synchronous method returns `void`, the corresponding asynchronous method returns `Task`. But if the synchronous method returns `TResult`, the corresponding asynchronous method will return `Task<TResult>`.
- You should also note that Microsoft suggests you consider the `async void` only for the event handlers (see https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios). Any other users do not follow the TAP model. In addition, the `async void` methods are difficult to test and any exception raised by them cannot be caught outside of that method.
- The TAP-based asynchronous method can perform a small number of synchronous operations, but it should return to the caller as quickly as possible. This is not a problem for you. It is because you are familiar with

the `await` keyword now. However, keep in mind that this point is very important when you work with I/O-bound tasks.

- The parameters of an asynchronous method and its corresponding synchronous method should match. These parameters also follow the same order.
- There are some exceptional cases when the previous rule is not applicable. For example, this rule does not hold for `ref` and `out` parameters. In fact, Microsoft suggests (see the detailed guideline link that is mentioned at the beginning of this section) to avoid them in your implementation by stating the following:

*Any data that would have been returned through an out or ref parameter should instead be returned as part of the **TResult** returned by **Task<TResult>**, and should use a tuple or a custom data structure to accommodate multiple values.*

- Providing progress notifications and/or cancellation tokens is optional for you. But Microsoft suggests you consider adding a `CancellationToken` parameter for the TAP method even if the corresponding synchronous method does not have such a parameter.

## Case Study

To bring more clarity to the topic, let's do some hands-on exercises. Suppose, someone has written a program where the program prints some numbers sequentially by calling a simple method, named `PrintNumbers`. You understand that a real-world scenario may deal with complex methods, which are time-consuming as well. To mimic the situation, I simulated some delay inside this method.

Here is the program:

```
using static System.Console;

var printTask = Task.Run(() => PrintNumbers(10));
try
{
    WriteLine($"Printed up to the number: {printTask.Result}");
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
        WriteLine($"Encountered error: {ae.Message}");
    }
}

WriteLine("End of the main thread.");

// The following method is synchronous.

static int PrintNumbers(int limit)
{
    int currentNumber = 0;
    WriteLine("The printing task starts now.");

    for (int i = 0; i < limit; i++)
    {
        Write($"PrintNumbers prints {i}\n");
        currentNumber = i;
        // Simulating some delay for some other activities, if any
        Thread.Sleep(500);
    }
    return currentNumber;
}
```

**Note** You can download the project `Demo_WithoutTAP` to execute this program.

You should not be surprised that you are expected to see the following output:

```
The printing task starts now.
PrintNumbers prints 0
PrintNumbers prints 1
PrintNumbers prints 2
PrintNumbers prints 3
PrintNumbers prints 4
PrintNumbers prints 5
PrintNumbers prints 6
PrintNumbers prints 7
PrintNumbers prints 8
PrintNumbers prints 9
Printed up to the number: 9
End of the main thread.
```

Though the program is simple, the problem is obvious. Now we are considering only 10 elements, but if we need to deal with thousands of elements, the execution time is longer. Also, you may need to cancel the program due to some typical reasons. So, your job is to make an asynchronous version of the method that supports cancellations as well. Can you do this?

I am sure that you can make an application using the concepts of tasks that you learned at the beginning of the book. **However, this time, I want you to make an asynchronous version of the `PrintNumber` method and follow the TAP pattern.** Can you make it? Let's search for a sample solution.

## Demonstration 6

In the upcoming program, you'll see the use of the following method:

```
static async Task<int> PrintNumbersAsync(int limit,
 CancellationToken token)
{
    int currentNumber = 0;
    WriteLine("The printing task starts now.");
    for (int i = 0; i < limit; i++)
    {
        token.ThrowIfCancellationRequested();
        Write($"PrintNumbersAsync prints {i}\n");
        currentNumber = i;
        // Simulating some delay
        await Task.Delay(500, token);
    }
    return currentNumber;
}
```

Let us verify whether this method follows the TAP pattern. You already see the use of the `async` and `await` keywords. In addition, we can confirm the following points:

- The method returns a `Task<TResult>` type. Here it is `Task<int>`.
- The method name has the `Async` suffix.
- This method accepts the same parameter (i.e., limit) as its synchronous version does.
- In addition, it supports the cancellation activity.

---

**Note** You can handle the possible exceptions inside this task. However, I preferred to handle them inside the main thread.

---

You can see the criteria for following the TAP pattern is satisfied. Let's see the sample program that demonstrates the usage of `PrintNumbersAsync`. It is as follows:

```
using static System.Console;

WriteLine("Task-based asynchronous pattern (TAP)
 demonstration.");

var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

var printTask = Task.Run(() => PrintNumbersAsync(10, token));

WriteLine("Enter c to cancel the task.");
var input = ReadKey();
if (input.KeyChar.Equals('c'))
{
    WriteLine("\nTask cancellation requested.");
    tokenSource.Cancel();
}

try
{
    WriteLine($"Printed up to the number: {printTask.Result}");
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
    {
```

```
        WriteLine($"Encountered error: {e.Message}");
    }
}

WriteLine("End of the main thread.");

// The PrintNumbersAsync is placed here. It is not shown again
// to avoid the repetition.
```

---

**Note** You can download the Demo_TAP project to see and execute the complete program.

---

## Output

Here is the output I got when I initiated a cancellation request while the program was running:

```
Task-based asynchronous pattern (TAP) demonstration.
Enter c to cancel the task.
The printing task starts now.
PrintNumbersAsync prints 0
PrintNumbersAsync prints 1
PrintNumbersAsync prints 2
PrintNumbersAsync prints 3
c
Task cancellation requested.
Encountered error: A task was canceled.
End of the main thread.
```

## Q&A Session

**QApp1.6 Why should a TAP-based asynchronous method return to the caller as quickly as possible?**

Here are some specific reasons behind this:

- If the method spends too much time on a time-consuming synchronous operation, the responsiveness of the application is affected, and the ultimate benefit of using asynchronous programming is lost.
- Consider another typical scenario when you concurrently launch multiple asynchronous methods. In this case, if one of these tasks performs a time-consuming synchronous operation, other asynchronous operations are delayed. As a result, the overall benefit of concurrency is lost as well.

In fact, you can easily understand the importance of quick returns when you consider the out-of-process calls. I assume that you'll also agree that if a UI thread calls a web page or a database, it should not block itself for the invoked operation to be completed.

**QApp1.7 What happens if a consumer calls the `Start` method on a hot task?**

A hot task is an active task. So, calling the `Start` method on such a task causes an `InvalidOperationException` to occur. For example, if you write something like the following:

```
// The previous code is the same
var printTask = Task.Run(() => PrintNumbersAsync(10, token));
printTask.Start();
// The remaining code is the same
```

and execute the program again, you'll see the error in the output. Here is a sample:

```
// The previous lines in the output are not shown
System.InvalidOperationException: Start may not be called on a
 promise-style task.
//The remaining lines in the output are not shown
```

**QApp1.8 I can see that you have not shown the progress status of this application. Was it intentional? If not, how can I support progress reporting in a similar application?**

Support for cancellation and progress reporting are both optional for TAP. You can use none of them, any of them, or both. So, four possible overloads for a particular asynchronous method are possible. Since Microsoft suggests we consider the support for cancellation, I have shown an application that fulfils the minimum criterion (and recommendation).

**QApp1.9 When an async method does not have the await keyword, I see the following warning message: "CS1998 This async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a background thread." I'd like to know the reason behind this.**

Microsoft does not want you to do this. This is very inefficient because, in this case, for the async method, the C# compiler unnecessarily generates a state machine that is of no use.

**QApp1.10 I am interested in displaying the progress reporting. How can I support the concept in the previous demonstration?**

Normally, to support progress reporting, developers use the `IProgress<T>` interface. Usually, you pass it as a parameter to the asynchronous method and name the parameter `progress`. However, you can exercise a simple logic to incorporate the progress reporting in this program. For example, see the updated method with the key changes in bold:

```
static async Task<int> PrintNumbersAsync(int limit,
 CancellationToken token)
{
    int total = 0;
    WriteLine("The printing task starts now.");
    for (int i = 0; i < limit; i++)
    {
        token.ThrowIfCancellationRequested();
        #region Old code
        // Write($"PrintNumbersAsync prints {i}\n");
        #endregion

        #region New code to display progress
        Write($"PrintNumbersAsync prints {i}");
        Write($"\tCompleted {(i + 1) * 100 / limit}%\n");
        #endregion

        total = i;
        // Simulating some delay
        await Task.Delay(500, token);
    }
    return total;
}
```

Now execute the program again. Here is some sample output for you:

```
Task-based asynchronous pattern (TAP) demonstration.
Enter c to cancel the task.
The printing task starts now.
PrintNumbersAsync prints 0      Completed 10%
PrintNumbersAsync prints 1      Completed 20%
PrintNumbersAsync prints 2      Completed 30%
PrintNumbersAsync prints 3      Completed 40%
c
```

```
Task cancellation requested.
Encountered error: A task was canceled.
End of the main thread.
```

---

**Note** You can download the `Demo_TAP_WithProgressReporting` project to see and execute the complete program.

---

# What's Next?

Parallel programming is a vast topic, and it is almost impossible to cover everything in a book. Also, this book is already fat! So, before I finish this book, I'll provide a few more words about the next steps:

- Throughout the book, I used many different built-in functions. There are many overloads available for these methods. You should explore them as well.
- This book covers many synchronization constructs. Still, there are many more. Based on the needs of your application, you may want to know about others as well. There are plenty of online and offline materials to serve you in this context.
- You saw me using the built-in task-based combinator `Task.Run` throughout this book. I want you to explore other constructs such as `Task.FromResult`, `Task.WhenAll`, and `Task.WhenAny` as well. Particularly, you have seen the last two when I talked about deadlock avoidance. You can get help from https://learn.microsoft.com/en-us/dotnet/standard/

asynchronous-programming-patterns/consuming-the-task-based-asynchronous-pattern#using-the-built-in-task-based-combinators.

- The new features can make the coding easy. So, I suggest you keep track of the latest features. C# 12 included several interesting features, and you have already seen the usage of the collection expressions, optional parameters in lambda expressions, and private constructors in this book.

**POINT TO NOTE**

However, the key point is to understand that new features will keep evolving; whatever is new today will be outdated tomorrow. On the contrary, the activity that involves understanding the fundamentals or core constructs is long-lasting and evergreen. This is why this book puts more emphasis on helping you understand the essentials of parallel programming and how the important constructs are evolving.

# A Personal Appeal

Over the years, I have seen a general trend in my books. When you like the book, you send me messages, write nice emails, and motivate me with your kind words and suggestions. Unfortunately, many of them do not reach review platforms like Amazon and others. However, when the opposite happens, I see all the criticisms on those pages.

Undoubtedly, these criticisms help me to write better. But it is also helpful for me to know what you liked in the book. These constructive suggestions help me to keep that information in an updated edition of the book.

So, I have a request for you: you can always point out the improvement areas of this work, but at the same time, please let me know about the areas you liked. I am sure that you also know that it is always easy to criticize, but an artistic view and open mind are required to discover the true efforts that are associated with any kind of work.

Thank you and happy coding!

## APPENDIX B

# Recommended Reading

This appendix lists some useful resources including paid courses. I have gotten many new insights from them. I believe that they will be equally effective for you.

## Books

The following list includes two books. In this list, the first one is my favorite. The second one is quite old but still helpful.

- *C# 12 in a Nutshell* by Joseph Albahari (O'Reilly Media, 2023)
- *Pro .NET 4 Parallel Programming in C#* by Adam Freeman (Apress, 2010)

I have learned many things from Stephen Cleary's blogs/articles. However, I have not read his book on this topic yet. If interested, you may look at the following one as well:

- *Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming Second Edition* (O'Reilly Media, 2019)

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5

# Courses

The following list includes helpful online courses. The first one covers a wide number of topics whereas the other courses cover a limited number of topics. At the time of this writing, none of them is free. However, you may get a promotional discount occasionally on these courses.

- www.udemy.com/course/parallel-dotnet/learn/lecture/5645430#overview
- www.linkedin.com/learning/advanced-threading-in-c-sharp
- www.linkedin.com/learning/advanced-c-sharp-thread-safe-data-with-concurrent-collections

# Other Resources

In each chapter, you have seen various online resources in the discussions and the "Q&A Sessions." If interested, you can have a detailed look at those resources to learn more about them. However, I recommend that you start with the following one: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/.

## APPENDIX C

# Other Books by the Author

The following list includes my other Apress books:

- *Introducing Functional Programming Using C#* (Apress, 2023)
- *Simple and Efficient Programming in C# Second Edition* (Apress, 2022)
- *Test Your Skills in C# Programming* (Apress, 2022)
- *Java Design Patterns Third Edition* (Apress, 2022)
- *Simple and Efficient Programming in C#* (Apress, 2021)
- *Design Patterns in C# Second Edition* (Apress, 2020)
- *Getting Started with Advanced C#* (Apress, 2020)
- *Interactive Object-Oriented Programming in Java Second Edition* (Apress, 2019)
- *Java Design Patterns Second Edition* (Apress, 2019)
- *Design Patterns in C#* (Apress, 2018)

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5

- *Interactive C#* (Apress, 2017)
- *Interactive Object-Oriented Programming in Java* (Apress, 2016)
- *Java Design Patterns* (Apress, 2016)

The following list includes my non-Apress books:

- *Python Bookcamp* (Amazon, 2021)
- *Operating System: Computer Science Interview Series* (Createspace, 2014)

To learn more about these books, you can refer to https://amazon.com/author/vaskaran_sarcar or to https://link.springer.com/search?new-search=true&query=vaskaran+sarcar&content-type=book&dateFrom=&dateTo=&sortBy=newestFirst.

# Index

## A

AddBlackCarModels method, 180, 191
Add methods, 150–152, 191, 198, 199
AddNonBlackCarModels method, 180, 191
AddOrUpdate method, 204
AggregateException, 79–80, 84, 388, 389
Aggregation methods, 316, 318
Aggregation operations, 315, 316, 318
AsParallel method, 284, 321
AsTask, 383, 384, 395
Async and await keywords, 349, 392, 428
async method, 336–340, 346, 387
Async-await programming model, 359
Asynchronous construction
  async and await keywords, 363, 365
  await operator, 366
Asynchronous initialization, 365, 366, 370, 379
Asynchronous programming, 333, 358, 359, 383
  analyze, 342
  await keyword, 334, 335, 339, 346
  await operator, 340
  challenging activity, 334
  demonstration, 340
  documentation, 337
  fire-and-forget operation, 345
  long-running operation, 338
  naming convention, 336
  output, 341
  restrictions, 338
  return statement, 337
  return types, 336
  type, 336
  void return type, 345
  worker thread are running, 342
Asynchronous programming models (APMs), 428
AutoResetEvent class, 144, 154–156, 161
await expression, 335, 353, 387, 397
Await_Restrictions project, 348

© Vaskaran Sarcar 2024
V. Sarcar, *Parallel Programming with C# and .NET*,
https://doi.org/10.1007/979-8-8688-0488-5

## B

BlockingCollection<T> class, 162, 194–201
Bounded blocking collection, 195, 201
Break method, 235, 237, 238, 240, 268, 274
Built-in functions, 439

## C

Cancellation operations, 103, 105, 310
Cancellation scenario, 135, 311
CancellationToken, 56, 92, 102, 242, 245
C# compiler, 27, 364, 368, 437
Child tasks, 13, 14, 30, 62, 107
Coding features, 440
Coding styles, 265–266, 280, 297
Collection expressions, 25, 66, 440
Computing machines, 2
ConcurrentBag<T> class, 162, 164, 165, 189–197
Concurrent collections
  BlockingCollection<T> class, 194–201
  ConcurrentBag<T> class, 189–197
  ConcurrentDictionary<TKey, TValue> class, 201–210
  ConcurrentQueue<T> class, 182–185
  ConcurrentStack<T> class, 172–182
  IProducerConsumer Collection<T> interface, 164, 165, 167
  notable characteristics, 168–171
  parallel programming, 162
  System.Collections.Generic namespace, 162
ConcurrentDictionary <TKey,TValue> class, 162, 170, 201–210
ConcurrentQueue<T> class, 162, 164–168, 182, 195, 196
  analysis, 185
  code segment, 183
  demonstration, 184
  Enqueue method, 183
  FIFO collection, 182
  Push and TryPop methods, 183
  PushRange method, 175
  sample output, 184
  TryDequeue method, 183, 184
  TryPeek method, 183, 184
  TryPopRange method, 173, 184
ConcurrentStack<T> class, 162, 165, 167–169
  AddBlackCarModels method, 178
  addBlackCarModelsTask, 177
  AddNonBlackCarModels method, 178
  addNonBlackCar ModelsTask, 177

analysis, 182
Car class, 177
concurrent stack, 176
ConcurrentStack<Car> cars = new ConcurrentStack <Car>(), 177
demonstration, complete program, 179
LIFO collection, 172
PushRange method, 175
RemoveCarModels method, 179
sample output, 182
TryPeek method, 172
TryPop method, 172
TryPopRange method, 173, 174
Continuation tasks
simple continuation scenario, 21, 22
specialized continuation scenario
analysis, 25
Collection expressions, 25
ContinueWhenAll method, 23
ContinueWhenAny, 27
output, 24, 29
parameters, 23
program, 24, 28, 29
Q&A session, 26
task1/task2, 23
TPL, 20
CountdownEvent class, 144, 154, 411, 412, 414
Custom aggregation, 316–319
CustomExceptions namespace, 313

## D

Data parallelism, 405, 406
Declarative programming, 265, 279–280
Demo2_UsingLock, 150
Demo3_InterlockedOps, 153
Dictionary<TKey,TValue>, 162
Dictionary<TKey,TValue> Stack<T>, 201
Dispose method, 201

## E

Employee class, 59
Enqueue method, 166, 183
EnqueueSlow method, 166
Enumerable class, 265–267, 283
Enumerable.Range(0, count), 284
Event-based asynchronous patterns (EAPs), 428
EventResetMode class, 156
EventWaitHandle class, 156
ExceedsCustomLimit Exception, 306
Exception handling
multiple locations
InsufficientMemory Exception, 87, 88
InvokeTasks method, 88
output, 91
program, 89, 90

Exception handling (*cont.*)
single location
ae.InnerExceptions, 83
AggregateException
class, 84, 85
Database class, 81, 82
output, 83
program, 82, 83
Source property, 85–87
Exceptions
AggregateException, 79, 80
CheckUser method, 75
creation, 72, 73
error details, 76
InnnerException, 78
InvalidDataException, 137
output, 73
Product class, 71
settings, 74
System.AggregateException, 137
Task.Run, 76
task1.Wait(), 77
UnauthorizedAccess
Exception, 74
validateUser, 75
ExecuteLongTaskAsync() method, 349, 354–356
Extension method, 281, 285, 298, 301

## F

Fixed deposit (FD), 91
for loop, 220, 223, 230, 234, 254, 257
ForAll method, 226, 285, 306
foreach function, 220, 287, 297
foreach loop, 84, 219, 220, 222, 286
foreach method, 222, 238, 268
Functional programming (FP), 80, 124, 265, 280

## G, H

GetFlagValue method, 370, 374
GetHundredAsync method, 339, 340
GetStringAsync method, 393, 401
GetType method, 359

## I, J, K

IAsyncStateMachine interface, 351
IDisposable interface, 201
Immutable types, 124
Implicit Global Usings, 57, 125, 267
InitializeAsync method, 366, 372, 374
InputDomain, 229
Instantiation process, 375
asynchronous initialization, 379
construction process, 378
constructors, 376
Foo, 375
ReadKey, 378
value property, 379
Interlocked class, 144, 150–153, 213
Invoke, 228
I/O-Bound tasks, 117–118, 136

IProducerConsumer Collection<T> interface, 162, 164, 165, 167, 195, 196
IsCompleted, 196, 238

## L

Language-Integrated Query (LINQ), 279
  AsParallel extension, 282
  functional style of coding, 281
  PLINQ, 279, 281
  queries, 281
List<T>, 163, 193, 297
Lock-free programming, 150
Locking constructs, 144
Lock statements, 148, 153, 166, 171, 213, 414, 415
LowestBreakIteration, 238, 239

## M

Main thread, 31, 75, 159, 160, 212, 264, 343, 344
ManualResetEvent class, 144, 154, 159, 161
MaxDegreeOfParallelism, 241, 242, 245
Merging data
  approaches, 300
  foreach loop, 300
  PLINQ query, 301
  threads, 300
Microsoft documentation, 281
MoveNext() method, 352, 353, 355
Multicore machine, 295, 403, 404
Multicore processors, 403
Multiple cancellation tokens
  CreateLinkedTokenSource method, 108
  demonstration, 108
  emergency cancellation, 111
  program, 108, 110
  Visual Studio user, 111
Multithreaded environment, 71, 124, 143, 176, 234
Multithreading environment, 2, 154, 161
Mutex, 144, 414–415, 419

## N

Namespaces, 3, 8, 9, 57, 125
Nested tasks, 133–135, 140
  documentation, 30
  task instances, 29
  TaskContinuationOptions
    analysis, 36, 37
    AttachedToParent, 34
    ContinueWith method, 33
    demonstration, 35, 36, 40, 41
    members, 33
    output, 36, 42
    Q&A session, 38, 39
    status, 33
  TaskCreationOptions, 31, 32
  task instances, 30

.NET Framework 4, 2
.NET Framework classes, 53
Nonblocking synchronization constructs, 144

## O

Object-oriented programming (OOP), 67, 80, 279
Online courses, 444
Online resources, 444

## P

Parallel class, 221, 222, 226, 241, 405, 406
Parallel custom aggregation, 318–319, 325
ParallelEnumerable class, 266, 267, 283, 318, 321
Parallel execution
  case studies, 230–233
  parallel for loop, 230
Parallel.Foreach method, 221, 230, 234, 260
Parallel.For method, 221, 234, 235, 260
Parallel.For loop, 257
Parallel Framework Extensions (PFX), 403
  components, 404, 405
  parallel programming, 405
  partitioning strategies, 405
  Q&A session, 406
Parallel LINQ (PLINQ), 260, 261, 265, 266
  declarative programming, 280, 286
  Enumerable's Range method, 286
  equivalent code, 298
  force parallelism, 298
  immediate execution, 290
  imperative and declarative styles, 279
  nondefault options, 305
  parallelism, 284
  processing order, 288
  query, 292, 295, 296, 322
  style of programming, 279
  uses, 280
  WithExecutionMethod, 284
Parallel loops
  fine-tune, 254
    partition-local variables, 260
    thread-local variables, 256–259
  functional style, 220
  handling exceptions, 250
    repeatedTask, 251
    sample output, 253
  manage cancellations, 245
    ParallelOptions instance, 246
    repeatedTask, 246
    sample output, 248
    static System.Console, 246
    Wait(token) method, 249

parallel foreach, 221–223, 234, 271
parallel foreach loops, 231, 234, 271
parallel for loop, 223, 234
analysis, 226
demonstration, 224
sample output, 225
parallel Invoke loop, 226
demonstration, 227
output, 227
Q&A session, 228–230
parallelism, 220
ParallelLoopState, 235, 258
ParallelMergeOptions, 301
Parallel processing, 293
Parallel programming, 2, 403, 405, 439
Parallel query, 286, 291, 325, 329
ParallelQuery{TSource}, 266, 267
Parameters
Parallel.For, 234, 235
ParallelLoopState, 235
Break, 237, 238
IsCompleted, 238
LowestBreakIteration, 238, 239
stop, 235, 236
ParallelOptions, 241
analysis, 244, 245
CancellationToken, 242
demonstration, 243
MaxDegreeOfParallelism, 241, 242
sample output, 244
TaskScheduler, 241
Parent task, 29, 33, 133–135, 140
Partitioning strategies, 405–406
Partition-local variable, 260
Producer-consumer pattern, 163, 194, 302
Push method, 165, 186

## Q

Q&A_ParallelForWith CustomStep, 230
Queue<T>, 182

## R

RemoveCarModels method, 179
repeatedTask, 239, 246
Repository class, 363
Reset method, 160, 407
RetrieveSomeValue method, 2

## S

Sample class, 368, 375, 379, 380
Semaphore, 144, 422–424, 427
SemaphoreSlim, 144, 422–424
Set method, 156, 160, 161, 212
Signaling, 154–156, 213, 407, 410, 411
Signaling constructs, 144
Single-threaded environment, 2
SomeTaskAsync, 345

SortedDictionary<TKey, TValue>, 163
SortedList<TKey,TValue>, 163
Stack<T>, Queue<T>, 162
StartNew() method, 12, 13, 15, 93, 362
state.Break(), 238
state.Stop(), 238
Static method, 372
Stop method, 235, 237, 238, 240
Synchronization, 143
  constructs, 144
    locking constructs, 144
    nonblocking synchronization constructs, 144
    signaling constructs, 144
    simple blocking methods, 144
  erroneous program, 145
  interlocked classes, 150
  lock statements, 148–150
  multithreaded environment, 145–148
  signaling, AutoResetEvent class, 155–162
Synchronization constructs, 439
  CountdownEvent
    demonstration, 412, 413
    error message, 414
    methods, 414
    metro railway stations, 411
    next-generation product, 411, 412
    output, 413
    uses, 411
  ManualResetEvent, 407
    analysis, 409
    implementation, 407
    output, 407, 409
    Q&A session, 409
  ManualResetEventSlim
    demonstration, 410
    output, 410
    Q&A session, 410, 411
  mutex
    Account class, 415–417
    Debit method, 416
    definition, 414
    Demo4_Mutex project, 416
    documentation, 414, 418
    flow, 419
    launching application, 421, 422
    output, 418
    program, 419, 420
    ReleaseMutex method, 416
    types, 418
    WaitOne method, 419
  SemaphoreSlim
    constructors, 423
    doctor's appointment, 422, 424
    initialCount parameter, 423
    number of spaces, 423
    order, 423
    output, 426
    program, 424, 426

Q&A session, 427
Release method, 423
WaitOne method, 423
Sequential foreach loop, 231
Synchronous method, 1, 334, 384, 429
System.Collections.Concurrent namespace, 162–163
System.Collections.Generic namespace, 162, 210

## T

Task-based asynchronous (TAP) pattern, 428
guidelines, 428
key points, 428–430
output, 432
PrintNumbers method, 430, 432
PrintNumbersAsync method, 433–435
program, 431
Q&A session, 436–438
Task-based combinator, 439
task class, 15, 17, 93, 228, 383, 393
TaskCompletionSource class, 118
demonstration, 121
details, 118, 119
init accessor, 124
job class, 119, 120
output, 120
points, 119
program, 121–123
TrySetResult method, 120
TaskFactory.StartNew method, 7, 360
Task parallelism, 4, 222, 405, 406
Task Parallel Library (TPL), 2, 80, 334
built-in supports, 20
documentation, 4
importance, 3
namespaces, 3
Parallel class, 3
purpose, 4
scenarios, 4
Task programming, 61, 136, 306
Task.Run method, 7, 71, 79, 136, 361
Tasks, 1
Child tasks, 107
code segment, 5
create and execute
approaches, 7
code segment, 5, 6
delegate, 6
demonstration, 8
output, 11
program, 8–10
Q&A session, 12, 13
Thread.Sleep method, 6
deadlocks, 427
definition, 5
examples, 5
final states, 60, 112
parent task, 133–135, 141
passing and returning values
approaches, 16, 17

Tasks (*cont.*)
code, 14
constructors, 15
demonstration, 18, 19
execution, 14
output, 19
PrintNumbersVersion2 function, 16
Q&A session, 20
StartNew method, 15, 68, 69
Task class/Result property, 17
Visual Studio, 18
Tasks cancellation
CancellationToken instance, 93
cancellation tokens, 92
CancellationTokenSource, 107
CancellationTokenSource.Cancel method, 97
demonstration, 94–96
enterprise application, 101
final status, Faulted, 102, 103
guidelines, 94
if condition, 94
monitoring
demonstration, 105, 106
register, 103
WaitOne method, 104
online documentation, 97
OperationCanceled Exception class, 98, 101, 102
options, 97
RanToCompletion, 99
Run method, 93
shortening code, 100, 101
StartNew method, 93
task constructor, 92
task definition, 98, 99
ThrowIfCancellationRequested method, 100
token, 93
token2, 102
var keyword, 92
Wait() method, 96
TaskScheduler instance, 241, 242
Task.Wait, 144, 213, 427
ThreadLocal<T> class, 251, 262, 263
Thread-local variable, 256–258, 260
ThreadPool class, 2, 3
Thread-safe collections, 162, 163, 170, 194
Thread.Sleep, 6, 46, 144, 213
ThreadStaticAttribute class, 262
Time-consuming method, 294, 339
ToList method, 297
Traditional foreach loop, 286, 291
TryAdd method, 164–167, 169, 189, 194, 196, 197, 202, 204
TryLocalPop, 189
TryPeek method, 186
TryPop method, 173, 186, 210
TryRemove method, 204
TrySteal method, 186, 188
TryTake method, 164, 167, 187, 188, 194
TryXXX methods, 194

## U

UnauthorizedAccess
Exception, 74, 389
Unwrap method, 362
await, 362
Run method, 363

## V

ValueTask, 383–385, 395
Visual Studio, 12, 18, 47, 73, 78, 111, 155, 188
Visual Studio Magazine, 358
Volatile keyword, 144

## W, X, Y, Z

Waiting for task
async/await programming, 56
how do we wait
cancellation token, 52
delay methods, 46, 47, 56
ReadKey() method, 48
Sleep method, 45, 139
WaitAll method, 49
WaitAny method, 49, 50, 138
Wait method, 48, 49, 51
WhenAny method, 51, 52
Sleep method, 55, 56
SpinUntil method, 54
SpinWait method, 53
WaitAny method, 55
WhenAny method, 55
why do we wait
analysis, 44
demonstration, 43, 44
outputs, 44
Waiting mechanism, 43, 61
Wait() method, 249
cases, 114, 115
money transfer process, 112
program, 112, 114
try-catch blocks, 112
Wait(token) method, 249
output, 115–117
try block, 115
*vs.* Wait() method, 116, 117
WaitOne method, 104, 161, 410, 419, 423
Websites, 59, 60
WithDegreeOfParallelism, 298, 322, 326
WithExecutionMode, 298
WithMergeOptions
method, 301
Work-stealing algorithm, 18